AltovaXML 2009

User and Reference Manual

AltovaXML 2009 User & Reference Manual

Published: 2009

Table of Contents

Chapter 1

Introduction

1 Introduction

AltovaXML 2009 is an XML application package which contains the Altova XML Validator, Altova XSLT 1.0 Engine, Altova XSLT 2.0 Engine, and Altova XQuery 1.0 Engine. The package is available, free of charge, as a single installer file from the Altova website. AltovaXML can be used to validate XML documents, transform XML documents using XSLT stylesheets, and execute XQuery documents.

AltovaXML can be used from the command line, via a COM interface, in Java programs, and in .NET applications. This documentation describes the usage of AltovaXML in all these environments, and also lists implementation-specific aspects of the engines in the package.

1.1 Product Features

The main features of AltovaXML are as follows:

Package

- XML Validator, XSLT Engines, and XQuery Engine packaged as a single installer file.
- Installer file available for download from Altova website free-of-charge.
- Easy installation of executable files on Windows systems.

Command line

- Command line usage for validation, XSLT transformation, and XQuery execution.
- Validation of XML documents according to DTD and W3C XML Schema rules.
- Transformation of XML documents with XSLT 1.0 and XSLT 2.0 stylesheets in conformance with respective W3C specifications.
- Execution of XQuery 1.0 documents in conformance with W3C specifications.

COM interface

- Can be used via COM interface, and therefore with applications and scripting languages that support COM.
- COM interface support is implemented for Raw and Dispatch interfaces.
- Wide range of XML validation, XSLT transformation, and XQuery execution features are available through interface properties.
- XML, DTD, XML Schema, XSLT, and XQuery input can be provided as files or as text strings in scripts and in application data.

Java interface

- AltovaXML functionality is available as Java classes that can be used in Java programs.
- Java classes provide XML validation, XSLT transformation, and XQuery execution features.

.NET interface

- A DLL file is built as a wrapper around AltovaXML and allows .NET users to connect to the functionality of AltovaXML.
- Provides primary interop assembly signed by Altova.
- Wide range of XML validation, XSLT transformation, and XQuery execution features are available.
- XML, DTD, XML Schema, XSLT, and XQuery input can be provided as files or as text strings in scripts and in application data.

1.2 Available Functionality

AltovaXML provides the functionality listed below. Most of this functionality is common to command line usage and COM interface usage. One major difference is that COM interface usage allows documents to be constructed from text strings via the application or scripting code (instead of referencing XML, DTD, XML Schema, XSLT, or XQuery files).

XML and XBRL Validation

- Validates the supplied XML document, returning valid or invalid.
- Validation can be done against the DTD or XML Schema referenced within the XML file, or against an external DTD or XML Schema supplied by a command line parameter or a COM interface property.
- Checks well-formedness of the supplied XML document, separately from validation.
- Validates XBRL documents. The XBRL document is validated against an XBRL taxonomy (which is a `.xsd` file) according to the rules of XBRL.

XSLT Transformations

- Transforms supplied XML document using supplied XSLT 1.0 or XSLT 2.0 document.
- XML document can be provided as a file via the input of a URL. In the case of usage via the COM interface, the XML document can alternatively be supplied as a text string.
- XSLT document can be provided as a file via the input of a URL. In the case of usage via the COM interface, the XSLT document can alternatively be supplied as a text string.
- Returns output documents at the named location. When called via COM interface can also return output documents as a string.
- XSLT parameters can be supplied via the command line and via the COM interface.

XQuery Execution

- Executes the supplied XQuery 1.0 document, optionally against an XML document named in a command line parameter or a COM interface property.
- XQuery document can be provided as a file via the input of a URL. In the case of usage via the COM interface, the XQuery document can alternatively be supplied as a text string.
- XML document can be provided as a file via the input of a URL. In the case of usage via the COM interface, the XML document can alternatively be supplied as a text string.
- Returns output documents at the named location. When called via COM interface can also return output documents as a string.
- External XQuery variables can be supplied via the command line and via the COM interface.
- Serialization options include: output encoding, output method (that is, whether the output is XML, XHTML, HTML, or Text), omitting the XML declaration, and indentation.

1.3 System Requirements and Installation

System requirements
AltovaXML is supported on Windows NT, Windows 2000, Windows XP, Windows Server 2003, and Windows Server 2008. To use AltovaXML via a COM interface, users should have privileges to use the COM interface, that is, to register the application and execute the relevant applications and/or scripts.

Installation
AltovaXML is available on the Altova website as a self-extracting download that will install AltovaXML with the necessary registrations. After you have downloaded the installer file (`AltovaXML2009.exe`) to your machine, double-click it to start the installation. The installer will install AltovaXML in the `Altova/AltovaXML2009` folder in the `Program Files` folder. All the necessary registrations to use AltovaXML via a COM interface, as a Java interface, and in the .NET environment will be done by the installer. This includes registering the AltovaXML executable as a COM server object, installing `AltovaXMLLib.dll` (for Java interface usage) in the `WINDIR\system32\` directory, and adding the `Altova.AltovaXML.dll` file to the .NET reference library.

You should note the following:

- For command line usage, invoke the installed executable file (`AltovaXML.exe`). This file can be copied to another accessible location on your machine or network and invoked from there.
- You can straightaway use AltovaXML via COM interface since the installed executable file `AltovaXML_COM.exe` will have been registered as a COM server object. If you change the location of the executable file `AltovaXML_COM.exe` to another location on your machine or to a mapped network drive, then you must manually register it at its new location as a COM server object. How to do this described in the section, Registering AltovaXML as a COM server object.
- In order to use AltovaXML via a Java interface, `AltovaXML_COM.exe` must be registered as a COM server object and the path to the file `AltovaXML.jar` (installed in the `Altova/AltovaXML2009` folder) must be added to the `CLASSPATH`. Registration as a COM server object is done automatically by the installer process. The installer also installs `AltovaXMLLib.dll` in the `WINDIR\system32\` directory. However, note that, if you change the location of the file `AltovaXML_COM.exe` after installation, then you must manually register it at its new location as a COM server object. See Registering AltovaXML as a COM Server Object and Java Interface for details.

1.4 About this Documentation

This documentation is the official product documentation of AltovaXML and provides comprehensive information about it. Its structure is as follows:

- The Introduction describes the features of the AltovaXML product, the functionality it provides, the main system requirements to use AltovaXML, and how AltovaXML is to be installed.
- The Usage section describes how to use AltovaXML from the command line and via a COM interface. The Command Line section provides details about the syntax used to invoke the various functionalities of AltovaXML. The COM Interface section describes how AltovaXML can be used with a COM interface; it provides a detailed description of the object model, its interfaces, and the properties of interfaces. The Java Interface section describes how AltovaXML can be used with Java and lists the defined Java interfaces and classes. The .NET Interface section provides a description of usage and lists the various methods and properties that can be used.
- The Engine Information section describes implementation-specific aspects of the various engines that are components of AltovaXML. Each engine is described separately.

Chapter 2

Usage

2 Usage

After AltovaXML has been downloaded and installed at the desired location, you can use it in the following ways:

- By calling the application from the command line,
- By using the application via a COM interface,
- By using the application via a Java interface, and
- By using the application in the .NET environment.

2.1 Command Line

To use AltovaXML from the command line, the executable file (`AltovaXML.exe`) must be installed/copied to an accessible location on your machine or network. The general syntax to call the application is:

```
AltovaXML functionality arg1 ... argN [options]
```

where

`AltovaXML`	Calls the application.
`functionality`	Specifies whether the XML validation, well-formedness check, XSLT 1.0 transformation, XSLT 2.0 transformation, or XQuery 1.0 execution functionality is called. Respective values are `-validate` (or `-v`), `-wellformed` (or `-w`), `-xslt1`,`-xslt2`, `-xquery` (or `-xq`).
`arg1 ... argN`	The arguments of the called functionality.
`options`	Each functionality has its own set of options. These are described in the corresponding sub-sections of this section.

General options

`-help`, `-h`, or `-?`	Displays usage information, i.e. a list of all arguments and options.
`-version`, `-ver`	Displays the program version.

The following functionality is available, and the allowed arguments and options for each functionality are described in detail in the corresponding sections:

- XML Validation and Well-Formedness
- XSLT 1.0 Transformations
- XSLT 2.0 Transformations
- XQuery 1.0 Executions

Usage summary

Given below is a summary of command line usage. For details, refer to the respective sections.

Using Altova XML Validator

- `-validate <filename> [-schema <filename> | -dtd <filename>]`
- `-wellformed <filename>`

Using Altova XSLT 1.0 Engine

- `-xslt1 <filename> -in <filename> [-param name=value] [-out <filename>]`

Using Altova XSLT 2.0 Engine

- `-xslt2 <filename> -in <filename> [-param name=value] [-out <filename>]`

Using Altova XQuery 1.0 Engine

- `-xquery <filename> [-in <filename>] [-param name=value] [-out <filename>] [serialization options]`

Note: If the filename or the path to it contains a space, then the entire path should be enclosed in quotes. For example: `"c:\My Files\MyXML.xml"` or `"c:\MyFiles\My XML.xml"`.

2.1.1 XML Validation and Well-Formedness

Syntax
The syntax to invoke **XML validation** is:

```
AltovaXML -validate xmlfile [-schema schemafile | -dtd dtdfile]
```

where

`AltovaXML`	Calls the application
`-validate` (or `-v`)	Specifies that the Altova XML Validator is to be used to validate the file `xmlfile`.

The following options are available:

`-schema` (or `-s`)	Specifies the XML Schema file `schemafile` to be used for validation.
`-dtd` (or `-d`)	Specifies the DTD file `dtdfile` to be used for validation.

Note:

- When no XML Schema or DTD file is specified as a command line option, an XML Schema or DTD file must be specified in the XML document itself.
- If an XML Schema or DTD file is specified as a command line option **and** an XML Schema or DTD file is referenced in the XML file, then the file specified in the command line option is used for validation.
- If an XBRL instance document is validated, the XBRL taxonomy, which is a `.xsd` file, is looked up.

The syntax to invoke the **well-formedness check** is:

```
AltovaXML -wellformed xmlfile
```

where

`AltovaXML`	Calls the application
`-wellformed` (or `-w`)	Specifies that the Altova XML Validator is to be used to check the well-formedness of the file `xmlfile`.

Examples

- `AltovaXML -validate test.xml -schema testschema.xml`
- `AltovaXML -v test.xml -dtd testdtd.dtd`
- `AltovaXML -wellformed test.xml`
- `AltovaXML -w test.xml`

Note: For using Altova XML in batch commands, it is important to know the following:

- The return code of the last executed command is stored in the `errorlevel` variable, the value of which can be retrieved with a batch command such as `ECHO %errorlevel%`.
- The return codes are `0` = well-formed/valid; `1` = not well-formed/invalid.

2.1.2 XSLT 1.0 Transformations

Syntax
The syntax to invoke XSLT 1.0 transformations is:

```
AltovaXML -xslt1 xsltfile -in xmlfile [-out outputfile] [options]
```

where

`AltovaXML`	Calls the application.
`-xslt1`	Specifies that the Altova XSLT 1.0 Engine is to be used for an XSLT transformation; the engine uses the XSLT 1.0 file *`xsltfile`* for the transformation.
`-in`	Specifies the XML file `xmlfile` to be transformed and its location.
`-out`	Specifies the output file `outputfile` and its location. If this option is omitted, the output is written to standard output.

The following options are available:

`-param`	Takes the instruction `paramname=XPath expression`. The `-param` switch is used before each global parameter. Double quotes must be used if a space is included in an XPath expression—whether in a path expression itself or in a string literal in the expression. See examples.
`-xslstack`	The stack size is the maximum depth of executed instructions, and can be changed with the `-xslstack` value. The minimum allowed value is 100. The default stack size is 1000. If the stack size is exceeded during a transformation, an error is reported.
`-namedTemplate` (or `-n`)	Sets the initial named template. A space separates the argument from its value. Example: `-namedTemplate MyTemplate`
`-mode` (or `-m`)	Sets the initial template mode. A space separates the argument from its value. Example: `-mode MyMode`

Note:

- The XSLT file must be specified in the command line instruction; an XSLT file referenced in an `<?xml-stylesheet?>` processing instruction in the XML document is not automatically used.
- If the `-out` parameter is omitted, output is written to the standard output.

Examples

- `AltovaXML -xslt1 test.xslt -in test.xml -out testout.xml`
- `AltovaXML -xslt1 test.xslt -in test.xml -out testout.xml -param date=//node/@att1`
- `AltovaXML -xslt1 test.xslt -in test.xml -out testout.xml -param date="//node/@att1 | //node/@att2"`
- `AltovaXML -xslt1 test.xslt -in test.xml -out testout.xml -param date=node/@att1 -param title='stringwithoutspace'`
- `AltovaXML -xslt1 test.xslt -in test.xml -out testout.xml -param date=node/@att1 -param title="'string with spaces'"`

2.1.3 XSLT 2.0 Transformations

Syntax
The syntax to invoke XSLT 2.0 transformations is:

```
AltovaXML -xslt2 xsltfile -in xmlfile [-out outputfile] [options]
```

where

`AltovaXML`	Calls the application.
`-xslt2`	Specifies that the Altova XSLT 2.0 Engine is to be used for an XSLT transformation; the engine uses the XSLT 2.0 file *`xsltfile`* for the transformation.
`-in`	Specifies the XML file `xmlfile` to be transformed and its location.
`-out`	Specifies the output file `outputfile` and its location. If this option is omitted, the output is written to standard output.

The following options are available:

`-param`	Takes the instruction `paramname=XPath expression`. The `-param` switch is used before each global parameter. Double quotes must be used if a space is included in an XPath expression—whether in a path expression itself or in a string literal in the expression. See examples.
`-xslstack`	The stack size is the maximum depth of executed instructions, and can be changed with the `-xslstack` value. The minimum allowed value is 100. The default stack size is 1000. If the stack size is exceeded during a transformation, an error is reported.
`-namedTemplate` (or `-n`)	Sets the initial named template. A space separates the argument from its value. Example: `-namedTemplate MyTemplate`
`-mode` (or `-m`)	Sets the initial template mode. A space separates the argument from its value. Example: `-mode MyMode`

Note:

- The XSLT file must be specified in the command line instruction; an XSLT file referenced in an `<?xml-stylesheet?>` processing instruction in the XML document is not automatically used.
- If the `-out` parameter is omitted, output is written to the standard output.
- The XSLT 2.0 Engine can be used in its backward compatibility mode to process an XSLT 1.0 stylesheet. The output, however, could be different than that produced by the XSLT 1.0 Engine processing the same XSLT 1.0 stylesheet.

Examples

- `AltovaXML -xslt2 test.xslt -in test.xml -out testout.xml`
- `AltovaXML -xslt2 test.xslt -in test.xml -out testout.xml -param date=//node/@att1`
- `AltovaXML -xslt2 test.xslt -in test.xml -out testout.xml -param date="//node/@att1 | //node/@att2"`
- `AltovaXML -xslt2 test.xslt -in test.xml -out testout.xml -param date=node/@att1 -param title='stringwithoutspace'`
- `AltovaXML -xslt2 test.xslt -in test.xml -out testout.xml -param date=node/@att1 -param title="'string with spaces'"`

2.1.4 XQuery 1.0 Executions

Syntax
The syntax to invoke XQuery 1.0 executions is:

```
AltovaXML -xquery xqueryfile [-in inputXMLfile -out outputfile]
   [options]
```

where

`AltovaXML`	Calls the application.
`-xquery` (or `-xq`)	Specifies that the Altova XQuery 1.0 Engine is to be used for an XQuery execution of the file `xqueryfile`.
`-in`	Specifies the input XML file.
`-out`	Specifies the output file and its location. If this option is omitted, output is written to the standard output.

The following options are available:

`-var`	Specifies an external variable and its value. Takes the form `name=value`. Any number of external variables can be submitted, but each must be preceded by the `-var` keyword. Variable values must be strings that conform to the lexical form of the datatype as which the variable has been declared.
`-xparam`	Specifies an XQuery parameter name and the parameter's value. Takes the form `name=XPathExpression`. Use double quotes to enclose the XPath expression if the expression contains spaces. Use single quotes to delimit string literals in the XPath expression. Any number of parameters can be submitted, but each must be preceded by the `-xparam` keyword.
`-outputMethod` (or `-om`)	Serialization option to specify the type of output. Valid values are `xml`, `html`, `xhtml`, and `text`. Default is `xml`.
`-omitXMLDeclaration` (or `-od`)	Serialization option to specify whether the XML declaration should be omitted from the output or not. Valid values are `yes` and `no`. Default is `yes`.
`-outputIndent` (or `-oi`)	Serialization option to specify whether the output should be indented or not. Valid values are `yes` and `no`. Default is `no`.
`-outputEncoding` (or `-oe`)	Serialization option to specify the character set of the output. Valid values are names in the IANA character set registry. Default is `UTF-8`.

Note: If the `-out` parameter is omitted, output is written to the standard output.

Examples

- `AltovaXML -xquery testquery.xq -out testout.xml`
- `AltovaXML -xquery testquery.xq -in products.xml -out testout.xml -var company=Altova -var date=2006-01-01`
- `AltovaXML -xquery testquery.xq -out testout.xml -xparam source = " doc( 'c:\test\books.xml' )//book "`
- `AltovaXML -xquery testquery.xq -in products.xml -out`

```
testout.xml
-var company=Altova -omitXMLDeclaration no -oe ASCII
```

2.2 COM Interface

When registered as a COM server object, AltovaXML can be invoked from within applications and scripting languages that have programming support for COM calls. This is useful because it enables XML document validation, XSLT transformations (XSLT 1.0 and XSLT 2.0), and XQuery 1.0 document executions to be performed, by AltovaXML, from within a wide range of user applications.

To use AltovaXML with applications and scripting languages that have a COM interface, you must first register AltovaXML as a COM server object. How to do this is described in Registering AltovaXML as a COM server object.

The AltovaXML object model and its properties are described in the following sub-sections of this section. (Note that you can use both the Raw Interface and Dispatch Interface of COM. The Raw Interface is used for programming languages (such as C++).The Dispatch Interface is used for scripting languages (such as JavaScript) that do not allow passing parameters by reference.) You can therefore use AltovaXML with:

- Scripting languages such as JavaScript or any other scripting language that supports the COM interface.
- Programming languages such as C++ or any other that supports the COM interface.
- Java and .NET, for which interfaces are built as a wrapper, with classes being created around the COM interface.

This section on COM interface usage ends with a set of examples of how various functionalities of AltovaXML can be invoked from within a variety of user applications.

Examples
For examples additional to those in this section, see the example files in the `Examples` folder in the application folder.

2.2.1 Registering AltovaXML as a COM Server Object

When you install AltovaXML 2009, AltovaXML_COM.exe will automatically be registered as a COM server object. If you need to change the location of AltovaXML_COM.exe, it is best to de-install AltovaXML and then re-install it at the required location. In this way the necessary unregistration and registration are carried out by the installer process. If you copy AltovaXML_COM.exe to another machine, you must manually register AltovaXML at its new location as a COM server object. How to do this is explained below. This description assumes that AltovaXML has been successfully installed.

Manual registration
To register AltovaXML as a COM server object, do the following:

1. Copy AltovaXML_COM.exe to the required location. If this location is not on the local machine, map this location to a network folder.
2. Open a Windows Command Prompt window, or, from the Start menu, select **Run...**.
3. Register the application as a COM server object by using the /regserver parameter. For example, if AltovaXML_COM.exe is in the folder c:\AltovaXML, then key in:

   ```
   c:\AltovaXML\AltovaXML_COM.exe /regserver
   ```

   and press Enter.

Checking success of the registration
If the registration was successful, the Registry should contain the classes AltovaXML.Application and AltovaXML.Application.1. These two classes will typically be found under HKEY_LOCAL_MACHINE\SOFTWARE\Classes.

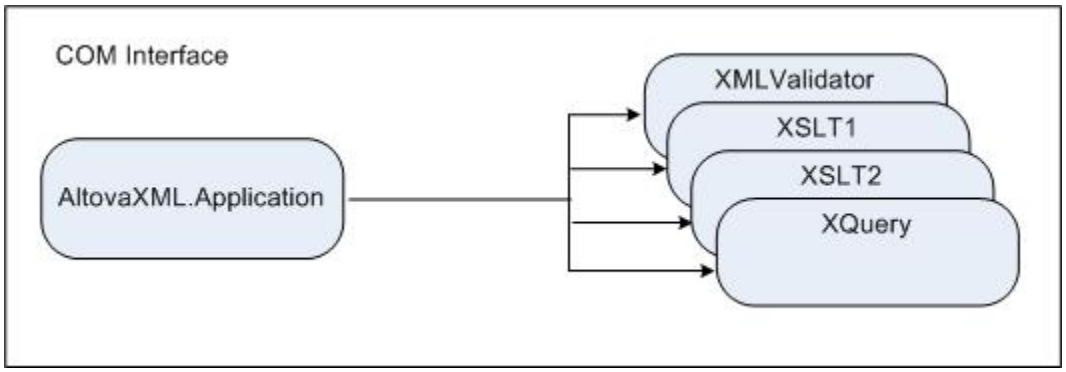
Manual unregistration
If the AltovaXML_COM.exe has been manually registered and you now wish to unregister it, then it should be manually unregistered. To manually unregister AltovaXML, call the application with the /unregserver parameter. For example, if the AltovaXML executable is in the folder c:\AltovaXML, then open a Windows Command Prompt window, key in c:\AltovaXML\AltovaXML_COM.exe /unregserver, and press Enter. You can check the Registry Editor for confirmation of unregistration.

Note: If AltovaXML was registered by the installer, the unregistration should be done by the installer—that is, by de-installing AltovaXML from the machine.

2.2.2 AltovaXML Object Model

The starting point for using the functionality of AltovaXML is the Application interface. This object contains the four objects that provide the AltovaXML functionality: XML validation, XSLT 1.0 transformations, XSLT 2.0 transformations, and XQuery 1.0 document processing. These objects have dual interfaces: the Dispatch Interface and the Raw Interface, which enables them to be used in scripting languages as well as in applications.

The object model of the AltovaXML API is depicted in the following diagram.



The hierarchy of the object model is shown below, and the five interfaces are described in detail in the corresponding sections. The properties and usage of each interface are described in the section for that interface.

- Application
  - XMLValidator
  - XSLT1
  - XSLT2
  - XQuery

Note:
Note the following general points about COM Interface usage:

- The term XML document refers not only to an XML document contained in an XML file but also to an XML document created with the `InputXMLFromText` property.
- Properties that take a resource location as its input accept absolute paths, as well as the HTTP and FTP protocols.
- When relative paths are used by a method to locate a resource, the resolution of the relative path should be defined in the calling module.

2.2.3 Application

Description

`AltovaXML.Application` is the root for all other objects. It is the only object you can create with the `CreateObject` function (of VisualBasic) or other similar COM-related functions.

Properties

`AltovaXML.Application` has the four properties listed below. Each of these functions returns the interface for the specific component. The details of each interface are given in the respective sections listed below.

- XMLValidator
- XSLT1
- XSLT2
- XQuery

Examples

Given below is a Visual Basic script that first creates the AltovaXML object, and then calls properties of the application interface.

```
Sub CommandButton1_Click()
Set objAltovaXML = CreateObject("AltovaXML.Application")

    objAltovaXML.XMLValidator.InputXMLFileName =
"c:\AltovaXML\test.xml"
    Sheet1.Cells(5, 2) = objAltovaXML.XMLValidator.IsValid

    objAltovaXML.XSLT1.InputXMLFromText = "<?xml version='1.0'
encoding='UTF-8'?><a><b/></a>"
    objAltovaXML.XSLT1.XSLFileName = "c:\workarea\altova_xml\1.xslt"
    Sheet1.Cells(6, 2) =
objAltovaXML.XSLT1.ExecuteAndGetResultAsString

End Sub
```

2.2.4 XMLValidator

Description
The `XMLValidator` interface provides methods to test:

- The well-formedness of an XML document.
- The validity of an XML document against a DTD or XML Schema referenced from within the XML document.
- The validity of an XML document against a DTD or XML Schema supplied externally via the code.
- The validity of an XBRL document against an XBRL taxonomy (a `.xsd` file).

All these methods return Boolean `TRUE` or `FALSE`. *See examples below*.

Note: Where string inputs are to be interpreted as URLs, absolute paths should be used. If a relative path is used, a mechanism to resolve the relative path should be defined in the calling module.

Methods
The following methods are available:

`IsWellFormed`
`IsWellFormed` checks the well-formedness of the XML document. Returns `TRUE` if the XML document is well-formed, `FALSE` if it is not well-formed.

`IsValid`
`IsValid` validates the XML document against the DTD or XML Schema referenced in the XML document. Returns `TRUE` if the XML document is valid, `FALSE` if invalid. To validate against a DTD or XML Schema not referenced in the XML document, use the method `IsValidWithExternalSchemaOrDTD`.

`IsValidWithExternalSchemaOrDTD`
`IsValidWithExternalSchemaOrDTD` validates the XML document against the DTD or XML Schema supplied by any one of the following properties: `SchemaFileName`, `DTDFileName`, `SchemaFromText`, or `DTDFromText`. If more than one of these properties has values set for it, then the `IsValidWithExternalSchemaOrDTD` method uses the property that has been set last. Returns `TRUE` if the XML document is valid, `FALSE` if invalid. To validate against a DTD or XML Schema referenced in the XML document, use the method `IsValid`.

Note: Validation and well-formedness checks must always occur after assigning the XML and/or DTD or XML Schema document to the respective properties.

Properties
The following properties are defined:

`InputXMLFileName`
A string input that is read as a URL to locate the XML file to be validated.

`SchemaFileName`
A string input that is read as a URL to locate the XML Schema file against which the XML document is to be validated.

`DTDFileName`
A string input that is read as a URL to locate the DTD file against which the XML document is to be validated.

InputXMLFromText
A string input that constructs an XML document.

SchemaFromText
A string input that constructs an XML Schema document.

DTDFromText
A string input that constructs a DTD document.

LastErrorMessage
Returns the last error message.

Examples

Given below is a single Visual Basic procedure that shows how the methods and properties of the `XMLValidator` interface can be used. This code is intended for use as a macro in an MS Excel worksheet, and references to worksheet cells indicate locations of input or output data. The file `c:\AltovaXML\test.xml` is assumed to contain a reference to a DTD.

```
Sub CommandButton1_Click()
Set objAltovaXML = CreateObject("AltovaXML.Application")

    objAltovaXML.XMLValidator.InputXMLFromText = "<?xml version='1.0'
encoding='UTF-8'?><a><b/></a>"
    Sheet1.Cells(4, 2) = objAltovaXML.XMLValidator.IsWellFormed

    objAltovaXML.XMLValidator.InputXMLFileName =
"c:\AltovaXML\test.xml"
    Sheet1.Cells(5, 2) = objAltovaXML.XMLValidator.IsValid

    objAltovaXML.XMLValidator.InputXMLFileName =
"c:\AltovaXML\test.xml"
    objAltovaXML.XMLValidator.DTDFileName = "c:\AltovaXML\test.dtd"
    Sheet1.Cells(6, 2) =
objAltovaXML.XMLValidator.IsValidWithExternalSchemaOrDTD

    objAltovaXML.XMLValidator.InputXMLFromText = "<?xml version='1.0'
encoding='UTF-8'?><a><b/></a>"
    objAltovaXML.XMLValidator.DTDFileName = "c:\AltovaXML\test.dtd"
    Sheet1.Cells(7, 2) =
objAltovaXML.XMLValidator.IsValidWithExternalSchemaOrDTD
End Sub
```

2.2.5 XSLT1

Description
The `XSLT1` interface provides methods and properties to execute an XSLT 1.0 transformation using the Altova XSLT 1.0 Engine. Results can be saved to a file or returned as a string. The interface also enables XSLT parameters to be passed to the XSLT stylesheet. The URLs of XML and XSLT files can be supplied as strings via interface properties. Alternatively, the XML and XSLT documents can be constructed within the scripting or programming code as text strings. *See examples below*.

Note: Where string inputs are to be interpreted as URLs, absolute paths should be used. If a relative path is used, a mechanism to resolve the relative path should be defined in the calling module.

Methods
The following methods are available:

`Execute`
`Execute` executes an XSLT 1.0 transformation and saves the result to an output file, the name and location of which is provided as an input string to the `Execute` method.

`ExecuteAndGetResultAsString`
`ExecuteAndGetResultAsString` executes an XSLT 1.0 transformation and returns the result as a UTF-16 text string.

`AddExternalParameter`
Takes a parameter name and the value of this parameter as input arguments. Each external parameter and its value is to be specified in a separate call to the method. Providing an external parameter with the name of an existing (uncleared) parameter causes an error. Since parameter values are XPath expressions, parameter values that are strings must be enclosed in single quotes. *See examples below*.

`ClearExternalParameterList`
No argument should be provided. The `ClearExternalParameterList` clears the external parameters list created with `AddExternalParameter` methods.

Note: Transformation must always occur after assigning the XML and XSLT documents.

Properties
The following properties are defined:

`InputXMLFileName`
A string input that is read as a URL to locate the XML file to be transformed.

`XSLFileName`
A string input that is read as a URL to locate the XSLT file to be used for the transformation.

`InputXMLFromText`
A string input that constructs an XML document.

`XSLFromText`
A string input that constructs an XSLT document.

`XSLStackSize`
The stack size is the maximum depth of executed instructions. The stack size can be changed

with the `XSLStackSize` property. The minimum allowed stack size is 100. The default stack size is 1000. If the stack size is exceeded during a transformation, an error is reported.

`LastErrorMessage`
Returns the last error message.

`JavaExtensionsEnabled`
Enables Java extensions. You can specify whether Java extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

`DotNetExtensionsEnabled`
Enables .NET extensions. You can specify whether .NET extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

Examples

Given below is a single Visual Basic procedure that shows how the various methods and properties of the `XSLT1` interface can be used. This code is intended for use as a macro in an MS Excel worksheet, and references to worksheet cells indicate locations of input or output data.

```
Sub CommandButton1_Click()
Set objAltovaXML = CreateObject("AltovaXML.Application")

    objAltovaXML.XSLT1.InputXMLFromText = "<?xml version='1.0'
encoding='UTF-8'?>
        <a><b/></a>"
    objAltovaXML.XSLT1.XSLFileName = "c:\AltovaXML\test.xslt"
    objAltovaXML.XSLT1.Execute "c:\AltovaXML\test_result.xml

    objAltovaXML.XSLT1.XSLStackSize = "500"
    objAltovaXML.XSLT1.InputXMLFromText = "<?xml version='1.0'
encoding='UTF-8'?>
        <company><name/><year>2005</year></company>"
    objAltovaXML.XSLT1.XSLFileName = "c:\AltovaXML\test.xslt"
    objAltovaXML.XSLT1.AddExternalParameter "web", "'www.altova.com'"
    objAltovaXML.XSLT1.AddExternalParameter "year", "/company/year"
    Sheet1.Cells(6, 2) =
objAltovaXML.XSLT1.ExecuteAndGetResultAsString
    objAltovaXML.XSLT1.ClearExternalParameterList
    objAltovaXML.XSLT1.AddExternalParameter "web",
"'www.nanonull.com'"
    objAltovaXML.XSLT1.AddExternalParameter "year", "/company/year"
    Sheet1.Cells(7, 2) =
objAltovaXML.XSLT1.ExecuteAndGetResultAsString
End Sub
```

2.2.6 XSLT2

Description
The `XSLT2` interface provides methods and properties to execute an XSLT 2.0 transformation using the Altova XSLT 2.0 Engine. Results can be saved to a file or returned as a string. The interface also enables XSLT parameters to be passed to the XSLT stylesheet. The URLs of XML and XSLT files can be supplied as strings via interface properties. Alternatively, the XML and XSLT documents can be constructed within the scripting or programming code as text strings. *See examples below*.

Note:

- Where string inputs are to be interpreted as URLs, absolute paths should be used. If a relative path is used, a mechanism to resolve the relative path should be defined in the calling module.
- The XSLT 2.0 Engine can be used in its backward compatibility mode to process an XSLT 1.0 stylesheet. The output, however, could be different than that produced by the XSLT 1.0 Engine processing the same XSLT 1.0 stylesheet.

Methods
The following methods are available:

`Execute`
`Execute` executes an XSLT 2.0 transformation and saves the result to an output file, the name and location of which is provided as an input string to the `Execute` method.

`ExecuteAndGetResultAsString`
`ExecuteAndGetResultAsString` executes an XSLT 2.0 transformation and returns the result as a UTF-16 text string.

`AddExternalParameter`
Takes a parameter name and the value of this parameter as input arguments. Each external parameter and its value is to be specified in a separate call to the method. Providing an external parameter with the name of an existing (uncleared) parameter causes an error. Since parameter values are XPath expressions, parameter values that are strings must be enclosed in single quotes. *See examples below*. Notice in the examples that the `date` parameter is given a value that is an XPath 2.0 function (`current-date()`).

`ClearExternalParameterList`
No argument should be provided. The `ClearExternalParameterList` clears the external parameters list created with `AddExternalParameter` methods.

`InitialTemplateName`
Sets the initial named template. The argument is the name of the template from which processing is to start.

`InitialTemplateMode`
Sets the initial mode for processing. The argument is the name of the required initial mode. Templates with this mode value will be processed.

Note: Transformation must always occur after assigning the XML and XSLT documents.

Properties
The following properties are defined:

InputXMLFileName
A string input that is read as a URL to locate the XML file to be transformed.

XSLFileName
A string input that is read as a URL to locate the XSLT file to be used for the transformation.

InputXMLFromText
A string input that constructs an XML document.

XSLFromText
A string input that constructs an XSLT document.

XSLStackSize
The stack size is the maximum depth of executed instructions. The stack size can be changed with the `XSLStackSize` property. The minimum allowed stack size is 100. The default stack size is 1000. If the stack size is exceeded during a transformation, an error is reported.

LastErrorMessage
Returns the last error message.

JavaExtensionsEnabled
Enables Java extensions. You can specify whether Java extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

DotNetExtensionsEnabled
Enables .NET extensions. You can specify whether .NET extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

Examples

Given below is a single Visual Basic procedure that shows how the various methods and properties of the `XSLT2` interface can be used. This code was intended for use as a macro in an MS Excel worksheet, and references to worksheet cells indicate locations of input or output data.

```
Sub CommandButton1_Click()
Set objAltovaXML = CreateObject("AltovaXML.Application")

    objAltovaXML.XSLT2.InputXMLFromText = "<?xml version='1.0'
encoding='UTF-8'?>
        <a><b/></a>"
    objAltovaXML.XSLT2.XSLFileName = "c:\AltovaXML\test.xslt"
    Sheet1.Cells(7, 2) =
objAltovaXML.XSLT2.ExecuteAndGetResultAsString

    objAltovaXML.XSLT2.XSLStackSize = "500"
    objAltovaXML.XSLT2.InputXMLFromText = "<?xml version='1.0'
encoding='UTF-8'?>
        <company><name/><year>2005</year></company>"
    objAltovaXML.XSLT2.XSLFileName = "c:\workarea\AltovaXML\2.xslt"
    objAltovaXML.XSLT2.AddExternalParameter "date", "current-date()"
    objAltovaXML.XSLT2.AddExternalParameter "hq", "'Vienna, Austria'"
    Sheet1.Cells(8, 2) =
objAltovaXML.XSLT2.ExecuteAndGetResultAsString
    objAltovaXML.XSLT2.AddExternalParameter "web",
"'www.nanonull.com'"
    objAltovaXML.XSLT2.AddExternalParameter "year", "/company/year"
    objAltovaXML.XSLT2.Execute
```



```
"c:\workarea\AltovaXML\test_result_xslt2.xml"
    Sheet1.Cells(9, 2) =
objAltovaXML.XSLT2.ExecuteAndGetResultAsString
End Sub
```

2.2.7 XQuery

Description
The `XQuery` interface provides methods and properties to execute an XQuery 1.0 transformation using the Altova XQuery 1.0 Engine. Results can be saved to a file or returned as a string. The interface also enables external XQuery variables to be passed to the XQuery document. The URLs of XQuery and XML files can be supplied as strings via interface properties. Alternatively, the XML and XQuery documents can be constructed within the scripting or programming code as text strings. *See examples below*.

Note: Where string inputs are to be interpreted as URLs, absolute paths should be used. If a relative path is used, a mechanism to resolve the relative path should be defined in the calling module.

Methods
The following methods are available:

`Execute`
`Execute` executes an XQuery 1.0 transformation and saves the result to an output file, the name and location of which is provided as an input string to the `Execute` method.

`ExecuteAndGetResultAsString`
`ExecuteAndGetResultAsString` executes an XQuery 1.0 transformation and returns the result as a UTF-16 text string.

`AddExternalVariable`
Takes a variable name and the value of this variable as input arguments. Each external variableand its value is to be specified in a separate call to the method. Variables must be declared in the XQuery document, optionally with a type declaration. Whatever the type declaration for the external variable in the XQuery document, the variable value submitted to the `AddExternalVariable` method does not need any special delimiter, such as quotes *(see example below)*. However, the lexical form must match that of the expected type (for example, a variable of type `xs:date` must have a value in the lexical form `2004-01-31`; a value in the lexical form `2004/Jan/01` will cause an error). Note that this also means that you cannot use an XQuery 1.0 function (for example, `current-date()`) as the value of an external variable (since the lexical form of the function as it is written will either not match the required data type (if the datatype is specified in the declaration of the external variable) or will be read as a string (if the datatype is not specified).) Providing an external variable that has the name of an existing (uncleared) variable causes an error.

`AddExternalVariableAsXPath`
Takes a variable name and the value of this variable as input arguments. Similar to `AddExternalVariable` method, except that `AddExternalVariableAsXPath` will be evaluated first as an XPath 2.0 expression. This makes it possible to pass in nodes and sequences with more than one element.

`ClearExternalVariableList`
No argument should be provided. The `ClearExternalVariableList` clears the external variables list created with `AddExternalVariable` methods.

Note: Setting the optional XML document must always be done before query execution.

Properties
The following properties are defined:

XQueryFileName
A string input that is read as a URL to locate the XQuery file to be executed. If both the `XQueryFileName` property and `XQueryFromText` property are specified, then the property that has been set later than the other (in the code sequence) is used.

InputXMLFileName
A string input that is read as a URL to locate the XML file that will be loaded into the query. XQuery navigation expressions are evaluated with reference to the document node of this XML document. If both the `InputXMLFileName` property and `InputXMLFromText` property are specified, then the property that has been set later than the other (in the code sequence) is used.

XQueryFromText
A string input that constructs an XQuery document. If both the `XQueryFileName` property and `XQueryFromText` property are specified, then the property that has been set later than the other (in the code sequence) is used.

InputXMLFromText
A string input that constructs an XML document. XQuery navigation expressions are evaluated with reference to the document node of this XML document. If both the `InputXMLFileName` property and `InputXMLFromText` property are specified, then the property that has been set later than the other (in the code sequence) is used.

LastErrorMessage
Returns the last error message.

JavaExtensionsEnabled
Enables Java extensions. You can specify whether Java extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

DotNetExtensionsEnabled
Enables .NET extensions. You can specify whether .NET extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

Note: If an XML document is set and is not needed for a new XQuery execution, then it should be cleared with an empty string assignment.

The following serialization options are defined:

OutputMethod
The required output method can be specified by submitting the required value as a string argument. Valid values are: `xml`, `xhtml`, `html`, and `text`. For example: `objAltovaXML.XQuery.OutputMethod = "xml"`. If the value is invalid, it is ignored. The default output method is `xml`.

OutputOmitXMLDeclaration
You can specify whether the XML declaration should be omitted or included in the output by submitting `true` or `false` (case-insensitive) as a Boolean argument. For example: `objAltovaXML.XQuery.OutputOmitXMLDeclaration = "FALSE"`. If the value is invalid, an error is raised. The default option is `TRUE`.

OutputIndent
You can specify whether the output should be indented or not by submitting `true` or `false` (case-insensitive) as a Boolean argument. For example: `objAltovaXML.XQuery.OutputIndent = "TRUE"`. If the value is invalid, an error is raised. The default option is False.

OutputEncoding
The required output encoding can be specified by submitting the encoding value as a string argument. For example: `objAltovaXML.XQuery.OutputEncoding = "UTF-8"`. If the value is invalid, it is ignored. The default output encoding is UTF-8.

Note: For the serialization options, Raw Interface and Dispatch Interface usage differs. In the Raw Interface, if no argument is provided with these properties, then the current value of the property is returned. You would use something like: `put_OutputOption( VARIANT_BOOL bVal )` or `VARIANT_BOOL bVal = get_OutputOption()`, respectively, to set values and get values. In the Dispatch Interface, you can use `b = myXQuery.OutputOption` to get values and `myXQuery.OutputOption = b` to set values. For example, in the Dispatch Interface, `Sheet1.Cells(10, 2) = objAltovaXML.XQuery.OutputEncoding` would get the current output encoding.

Examples
Given below is a single Visual Basic procedure that shows how the various methods and properties of the `XQuery` interface can be used. This code was intended for use as a macro in an MS Excel worksheet, and references to worksheet cells indicate locations of input or output data.

```
Sub CommandButton1_Click()
Set objAltovaXML = CreateObject("AltovaXML.Application")

    objAltovaXML.XQuery.InputXMLFileName = "c:\AltovaXML\test.xml"
    objAltovaXML.XQuery.XQueryFromText = " xquery version '1.0';
       declare variable $string as xs:string external;
       declare variable $num as xs:decimal external;
       declare variable $date as xs:date external;
       $string, ' ', 2*$num, ' ', $date "
    objAltovaXML.XQuery.AddExternalVariable "string", "A string"
    objAltovaXML.XQuery.AddExternalVariable "num", "2.1"
    objAltovaXML.XQuery.AddExternalVariable "date", "2005-04-21"
    Sheet1.Cells(10, 2) = objAltovaXML.XQuery.OutputEncoding
    objAltovaXML.XQuery.OutputMethod = "text"
    Sheet1.Cells(11, 2) = objAltovaXML.XQuery.OutputMethod
    objAltovaXML.XQuery.OutputIndent = "TRUE"
    Sheet1.Cells(12, 2) = objAltovaXML.XQuery.OutputIndent
    objAltovaXML.XQuery.OutputOmitXMLDeclaration = "FALSE"
    Sheet1.Cells(13, 2) = objAltovaXML.XQuery.OutputOmitXMLDeclaration
    Sheet1.Cells(14, 2) =
objAltovaXML.XQuery.ExecuteAndGetResultAsString
End Sub
```

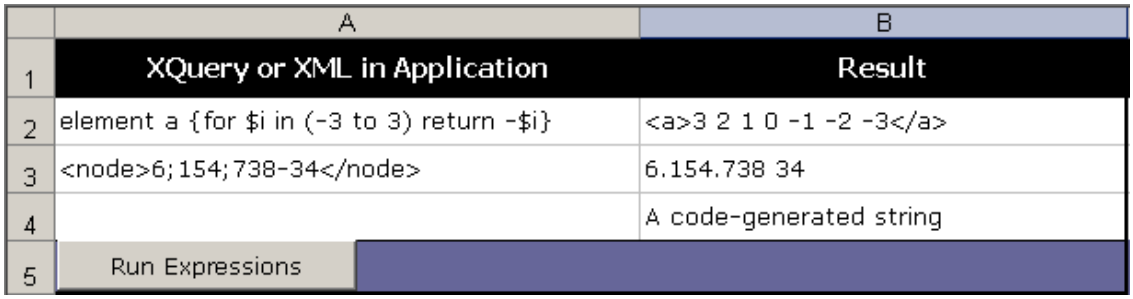
2.2.8 Examples

This section contains example code in (i) Visual Basic for an Excel macro; (ii) JScript; and (iii) C++. These examples will give you an idea of how you can use AltovaXML with a COM Interface.

For more detailed examples, see the example files in the `Examples` folder in the application folder.

Visual Basic

The following Visual Basic example is the code for a macro in an Excel worksheet (*screenshot below*). The macro has been assigned to the button `Run Expressions`. On clicking the button, the Visual Basic code is executed.



	A	B
1	XQuery or XML in Application	Result
2	element a {for $i in (-3 to 3) return -$i}	<a>3 2 1 0 -1 -2 -3
3	<node>6;154;738-34</node>	6.154.738 34
4		A code-generated string
5	Run Expressions	

Code sample

The Visual Basic code below uses the `XQuery` interface.

```
Sub CommandButton1_Click()
Set objAltovaXML = CreateObject("AltovaXML.Application")

    objAltovaXML.XQuery.XQueryFromText = Sheet1.Cells(2, 1)
    Sheet1.Cells(2, 2) =
objAltovaXML.XQuery.ExecuteAndGetResultAsString

    objAltovaXML.XQuery.InputXMLFromText = Sheet1.Cells(3, 1)
    objAltovaXML.XQuery.XQueryFromText = "translate(node, ';-', '. ')"
    Sheet1.Cells(3, 2) =
objAltovaXML.XQuery.ExecuteAndGetResultAsString

    objAltovaXML.XQuery.InputXMLFromText = "<a myAttr='A
code-generated string'/>"
    objAltovaXML.XQuery.XQueryFromText = "string(/a/@*)"
    Sheet1.Cells(4, 2) =
objAltovaXML.XQuery.ExecuteAndGetResultAsString
End Sub
```

On clicking the button **Run Expressions** in the Excel worksheet, the following three XQuery instructions are executed:

1. The input for the `XQueryFromText` property is an XQuery expression taken as text from the Excel worksheet cell 2A. The `ExecuteAndGetResultAsString` property executes the XQuery expression and places the result in the Excel worksheet cell 2B.
2. The input for the `InputXMLFromText` property is an XML fragment taken from the Excel worksheet cell 3A. The XQuery expression is given to the `XQueryFromText` property directly in the code. The result is placed in the Excel worksheet cell 3B.
3. The `InputXMLFromText` property creates an XML tree from the XML fragment

provided to it. The XQuery expression is given to the `XQueryFromText` property directly in the code, and the result is placed in the Excel worksheet cell 4B.

JScript

Given below is a JScript code sample that shows how AltovaXML can be used via the COM interface.

Code sample

```
// ////////////// global variables /////////////////
var objAltovaXML = null;

// ////////////////////// Helpers ///////////////////////////////

function Exit(strErrorText)
{
   WScript.Echo(strErrorText);

   if (objAltovaXML != null)
      objAltovaXML.Quit();

   WScript.Quit(-1);
}

function ERROR(strText, objErr)
{
   if (objErr != null)
      Exit ("ERROR: (" + (objErr.number & 0xffff) + ")" + objErr.description +
" - " + strText);
   else
      Exit ("ERROR: " + strText);
}

function CreateGlobalObjects ()
{
   // create the AltovaXML connection
   // if there is a running instance of AltovaXML (that never had a
connection) - use it
   // otherwise, we automatically create a new instance
   try
   {
      objAltovaXML = WScript.GetObject("", "AltovaXML.Application");
      //WScript.Echo("Successfully accessing AltovaXML.Application");
   }
   catch(err)
   {
   WScript.Echo(err)
      { Exit("Can't access or create AltovaXML.Application"); }
   }
}


// ////////////////////// MAIN ///////////////////////////////

CreateGlobalObjects();

objAltovaXML.XQuery.InputXMLFromText = " \
  <bib> \
  <book year=\"1994\">  \
     <title>TCP/IP Illustrated</title>  \
     <author><last>Stevens</last><first>W.</first></author>  \
     <publisher>AW</publisher>  \
     <price>65.95</price>  \
```



```
  </book>  \
  <book year=\"1992\">  \
     <title>Advanced Programming in the Unix Environment</title>  \
     <author><last>Stevens</last><first>W. </first></author>  \
     <publisher>AW</publisher>  \
     <price>65.95</price> \
  </book>  \
  <book year=\"2000\">  \
     <title>Data on the Web</title>  \
     <author><last>Abiteboul</last><first>Serge</first></author>  \
     <author><last>Abiteboul</last><first>Serge</first></author>  \
     <author><last>Abiteboul</last><first>Serge</first></author>  \
     <publisher>John Jameson Publishers</publisher>  \
     <price>39.95</price>  \
  </book>  \
  <book year=\"1999\">  \
     <title>Digital TV</title>  \

<editor><last>Gassy</last><first>Viktor</first><affiliation>CITI</affiliation>
</editor>  \
     <publisher>Kingston Academic Press</publisher>  \
     <price>129.95</price>  \
  </book>  \
 </bib> ";


objAltovaXML.XQuery.XQueryFromText = "\
 (: Filename: xmpQ1.xq :)  \
 (: Source: http://www.w3.org/TR/xquery-use-cases/#xmp-data :)  \
 (: Section: 1.1.1.9 Q1 :)  \
 (: List books published by AW after 1991, including their year and title.:)
\
 <bib>  \
 { \
     for $b in /bib/book  where $b/publisher = \"AW\" and $b/@year > 1991   \
              return <book year=\"{ $b/@year }\">     { $b/title }    </book>
\
 } \
 </bib> ";


var sResult = objAltovaXML.XQuery.ExecuteAndGetResultAsString();
WScript.Echo(sResult);
```

C++

Given below is a C++ code sample that shows how AltovaXML can be used via the COM interface.

Code sample

```
// TestAltovaXML.cpp : Defines the entry point for the console application.
//
#include "objbase.h"
#include <iostream>
#include "atlbase.h"

#import "AltovaXML_COM.exe" no_namespace raw_interfaces_only
// - or -
//#import "AltovaXML_COM.exe" raw_interfaces_only
//using namespace AltovaXMLLib;

int main(int argc, char* argv[])
{
       HRESULT hr = S_OK;
```

```
	hr = CoInitialize( NULL);
	if ( hr == S_OK )
	{
		IApplicationPtr ipApplication;

		hr = CoCreateInstance(
								__uuidof( Application
),
								NULL,
								CLSCTX_ALL,
								__uuidof( IApplication),

reinterpret_cast<void**>( &ipApplication)
							);
		if ( hr == S_OK )
		{
			IXQueryPtr ipXQuery;
			hr = ipApplication->get_XQuery( &ipXQuery );

			if ( hr == S_OK )
			{
				CComBSTR sXQExpr( "(1 to 10)[. mod 2 != 0]" );
				BSTR bstrResult;

				hr = ipXQuery->put_XQueryFromText( sXQExpr );
				hr = ipXQuery->ExecuteAndGetResultAsString(
&bstrResult );

				std::cout << (char*) _bstr_t( bstrResult) <<
std::endl;

				ipXQuery.Release();
			}

			ipApplication.Release();
		}

		CoUninitialize();
	}
	return 0;
}
```

2.3 Java Interface

The AltovaXML Java interface (AltovaXML.jar) connects to the AltovaXML COM interface using native functions in the AltovaXMLLib.dll. This DLL will have been installed in the WINDIR\system32\ directory when you install AltovaXML using the AltovaXML installer. AltovaXML.jar contains the package com.altova.engines, which is the package containing the Altova engines.

Setup

In order to use the Java interface, add the AltovaXML.jar file to the CLASSPATH. COM registration is done automatically by the AltovaXML Installer. If you change the location of the file AltovaXML_COM.exe after installation, you should register AltovaXML as a COM server object by running the command AltovaXML_COM.exe /regserver. See Registering AltovaXML as a COM Server Object for more details.

Documentation

This section contains a detailed description of the AltovaXML Java interface. This documentation is also available in HTML format in the ZIP archive, AltovaXMLJavaDocs.zip , which is located in the AltovaXML2009 application folder.

Examples

For detailed examples, see the example files in the Examples folder in the application folder.

The com.altova.engines package

To use the Java interface, your starting point is the package com.altova.engines. This is the Java interface for the AltovaXML COM server object; it provides access to XMLValidator and to the XSLT 1.0, XSLT 2.0 and XQuery 1.0 engines.

The com.altova.engines package provides connection to the AltovaXML COM interface using the native functions in AltovaXMLLib.dll, which is installed in the WINDIR\system32\ directory.

To connect to a new instance of AltovaXML COM server object, use the static method getInstance() of the AltovaXMLFactory class. From the returned interface you can choose the required engine using the getENGINENAMEInstance() function.

Given below is a sample of code that uses the Java interface:

```
import com.altova.engines.*;

/**
 * Test application for AltovaXML COM components java interface
 */
public class AltovaXMLTest {
    /**
     * public constructor for AltovaXMLTest
     */
    public AltovaXMLTest(){
    }

    /**
     * application main
     */
    public static void main(String[] args) {
```

```
        System.out.println("AltovaXML Java Interface Test Application");

        //request a COM server object - fails if AltovaXML is not registered
        IAltovaXMLFactory objXmlApp = AltovaXMLFactory.getInstance();

        if ( objXmlApp != null ) {
            //get interface for the XQuery engine
            IXQuery xquery = objXmlApp.getXQueryInstance();
            //set XQuery statement
            xquery.setXQueryStatement("<doc><a>{1 to 3}</a>This data is
well-formed.</doc>");
            //execute the statement previously set.
            //There was no input XML specified so the initial context is
empty.
            String sres = xquery.executeAndGetResultAsString();
            //release XQuery engine's connection to the COM server object
            xquery.releaseInstance();
            System.out.println(sres);


            IXMLValidator validator = objXmlApp.getXMLValidatorInstance();
            validator.setInputXMLFromText(sres);
            boolean b = validator.isWellFormed();
            if ( b )
                System.out.println("XML data is well-formed.");
            else
                System.out.println("Data is not well-formed.");
            validator.releaseInstance();

            //release Application object connection to the COM server object.
            //After this the COM server object will shut down automatically.
            objXmlApp.releaseInstance();
        } else{
            System.out.println("Creating instance of IAltovaXMLFactory
failed.");
            System.out.println("Please make sure AltovaXML.exe is correctly
registered!");
        }
    }
}
```


2.3.1 Interfaces

Given below is a summary of the interfaces of `com.altova.engines`. Detailed descriptions are given in the respective sections.

- `IAltovaXMLEngine`
  Basic interface for XMLValidator, and XSLT 1.0, XSLT 2.0, and XQuery 1.0 engines.
- `IAltovaXMLFactory`
  Interface for AltovaXML COM object wrapper.
- `IExecutable`
  Executable interface for engines.
- `IReleasable`
  Interface for Release functionality.
- `IXMLValidator`
  Interface for XMLValidator.
- `IXQuery`
  Interface for the XQuery 1.0 engine.
- `IXSLT`
  Interface for the XSLT engines.

IAltovaXMLEngine

Basic interface for XMLValidator, XSLT 1.0, XSLT 2.0 and XQuery engines. Public interface that extends IReleasable.

Superinterface: `IReleasable`
Subinterface: `XMLValidator`, `IXQuery`, `IXSLT`
Implementing classes: `XMLValidator`, `XQuery`, `XSLT1`, `XSLT2`

Methods
The following methods are defined.

`setInputXMLFileName`
`public void setInputXMLFileName( java.lang.String filename)`
Sets the file name for the input XML data. Please note that you have to use absolute URLs.
Parameters:
`filename`: an absolute URL giving the base location of the XML data.

`setInputXMLFromText`
`public void setInputXMLFromText( java.lang.String text)`
Sets the text value for the input XML data. For example: `setInputXMLFromText( "<doc> <a>text</a> </doc>" )`
Parameters:
`text`: a string containing XML data.

`getLastErrorMessage`
`public java.lang.String getLastErrorMessage()`
Gets the last error message from the engine.
Returns:
a string containing the last error message.

IAltovaXMLFactory

Interface for AltovaXML COM object wrapper. Provides access to the interfaces of XMLValidator, XSLT 1.0, XSLT 2.0 and Xquery 1.0 engines. Public interface that extends IReleasable.

Superinterface: IReleasable
Implementing classes: AltovaXMLFactory

Methods
The following methods are defined.

getXQueryInstance
public IXQuery **getXQueryInstance**( )
Creates a new instance of XQuery class for the current XQuery engine instance. The object's connection to the engine must be released after use. To do this, use the function releaseInstance( ) declared in the IReleasable interface.
Returns:
the IXQuery interface of the newly created class.

getXSLT1Instance
public IXSLT **getXSLT1Instance**( )
Creates a new instance of XSLT1 class for the current XSLT 1.0 engine instance. The object's connection to the engine must be released after use. To do this, use the function releaseInstance( ) declared in the IReleasable interface.
Returns:
the IXSLT interface of the newly created class.

getXSLT2Instance
public IXSLT **getXSLT2Instance**( )
Creates a new instance of XSLT2 class for the current XSLT 2.0 engine instance. The object's connection to the engine must be released after use. To do this, use the function releaseInstance( ) declared in the IReleasable interface.
Returns:
the IXSLT interface of the newly created class.

getXMLValidatorInstance
public IXMLValidator **getXMLValidatorInstance**( )
Creates a new instance of XMLValidator class for the current XML Validator instance. The object's connection to the engine must be released after use. To do this, use the function releaseInstance( ) declared in the IReleasable interface.
Returns:
the IXMLValidator interface of the newly created class.

IExecutable

Executable interface for engines. Public interface.

Subinterface: IXQuery, IXSLT
Implementing classes: XQuery, XSLT1, XSLT2

Methods

The following methods are defined.

execute

```
public boolean execute( java.lang.String outfilename)
```

Executes and saves the result to file. In case of an error, you can use the function getLastErrorMessage() declared in IAltovaXMLEngine to get additional information.

Parameters:

outfilename: an absolute URL giving the location of the output file.

Returns:

true on success, false on error.

executeAndGetResultAsString

```
public java.lang.String executeAndGetResultAsString()
```

Executes and returns the result as string. In case of an error, you can use the function getLastErrorMessage() declared in IAltovaXMLEngine to get additional information.

Returns:

string containing the serialized result. On error, will return the empty string.

enableJavaExtensions

```
public void enableJavaExtensions( boolean bEnable)
```

Enables/disables .NET extension functions.

enableDotNetExtensions

```
public void enableDotNetExtensions( boolean bEnable)
```

Enables/disables Java extension functions.

IReleasable

Public interface for Release functionality. When an object implementing this interface is not used any more, then the releaseInstance() function must be called in order to release connection to the COM server. The COM server will shut down automatically when all connections to it are released.

Subinterface: IXQuery, IXSLT
Implementing classes: XQuery, XSLT1, XSLT2

Methods

The following methods are defined.

releaseInstance

```
public void releaseInstance()
```

Releases the object's connection to the COM server.

IXMLValidator

Interface for the XML Validator. Public interface that extends IAltovaXMLEngine.

Superinterface: IAltovaXMLEngine, IReleasable
Implementing classes: XMLValidator

Methods

The following methods are defined.

isValid
public boolean **isValid**()
Validates the input XML data against the DTD/Schema specified in it.
Returns:
true on success, false on failure. In case of failure, you can use the function getLastErrorMessage() declared in IAltovaXMLEngine to get additional information.

isWellFormed
public boolean **isWellFormed**()
Checks the input XML data for well-formedness.
Returns:
true on success, false on failure. In case of failure, you can use the function getLastErrorMessage() declared in IAltovaXMLEngine to get additional information.

isValidWithExternalSchemaOrDTD
public boolean **isValidWithExternalSchemaOrDTD**()
Validates the input XML data against the external DTD/Schema which can be specified with the functions setDTDFileName(),setDTDFromText(),setSchemaFileName(), setSchemaFromText().
Returns:
true on success, false on failure. In case of failure, you can use the function getLastErrorMessage() declared in IAltovaXMLEngine to get additional information.

setSchemaFileName
public void **setSchemaFileName**(java.lang.String filename)
Sets file name for external Schema.
Parameters:
filename: an absolute URL giving the base location of the Schema

setDTDFileName
public void **setDTDFileName**(java.lang.String filename)
Sets file name for external DTD.
Parameters:
filename: an absolute URL giving the base location of the DTD.

setSchemaFromText
public void **setSchemaFromText**(java.lang.String text)
Sets text value for external Schema.

Parameters:
`text`: string containing Schema as text.

setDTDFromText
`public void setDTDFromText(java.lang.String text)`
Sets text value for external DTD.
Parameters:
`text`: string containing DTD as text.

IXQuery

Interface for the XQuery engine. Public interface that extends `IAltovaXMLEngine` and `IExecutable`.

Superinterface: `IAltovaXMLEngine`, `IExecutable`, `IReleasable`
Implementing classes: `XQuery`

Methods
The following methods are defined.

setXQueryFileName
`public void setXQueryFileName(java.lang.String filename)`
Sets the file name of the XQuery document.
Parameters:
`filename`: an absolute URL giving the base location of the XQuery file.

setXQueryStatement
`public void setXQueryStatement(java.lang.String text)`
Sets the text value of the XQuery statement.
Parameters:
`text`: a string containing the XQuery statement.

setOutputEncoding
`public void setOutputEncoding(java.lang.String encoding)`
Sets the encoding of the result document.
Parameters:
`encoding`: a string containing the name of the encoding name (for example: `UTF-8`, `UTF-16`, `ASCII`, `8859-1`, `1252`).

getOutputEncoding
`public java.lang.String getOutputEncoding()`
Retrieves the encoding specified for the result document.
Returns:
a string containing an encoding name.

setOutputIndent
`public void setOutputIndent(boolean indent)`
Enables/disables the indentation option for the result document.
Parameters:

indent: boolean value to enable/disable output indentation.

getOutputIndent
public boolean **getOutputIndent**()
Retrieves the output indent option specified for the result document.
Returns:
boolean value indicating whether output is indented (true) or not (false).

setOutputMethod
public void **setOutputMethod**(java.lang.String method)
Sets the serialization method for the result document.
Parameters:
method: a string containing the serialization method. (Valid values are: xml, xhtml, html, text).

getOutputMethod
public java.lang.String **getOutputMethod**()
Retrieves the serialization method for the result document.
Returns:
a string containing the serialization method for the output document.

setOutputOmitXMLDeclaration
public void **setOutputOmitXMLDeclaration**(boolean decl)
Enables/disables the serialization option omitXMLDeclaration for the result document.
Parameters:
decl: new boolean value for the omit-xml-declaration parameter.

getOutputOmitXMLDeclaration
public boolean **getOutputOmitXMLDeclaration**()
Retrieve the value of omitXMLDeclaration option specified for the result document.
Returns:
boolean value indicating whether output document contains an XML declaration (true) or not (false).

addExternalVariable
public void **addExternalVariable**(java.lang.String name,
java.lang.String val)
Add name and value for an external variable.
Parameters:
name: a string containing a valid QName as the variable name.
val: a string containing the value of the variable; the value will be used as a string.

addExternalVariableAsXPath
public void **addExternalVariableAsXPath**(java.lang.String name,
java.lang.String val)
Add name and value for an external variable, with value being evaluated as an XPath 2.0 expression.
Parameters:
name: a string containing a valid QName as the variable name.

val: a string containing the value of the variable; the value will be evaluated as an XPath 2.0 expression.

clearExternalVariableList
public void **clearExternalVariableList**()
Clears the list of external variables.

IXSLT

Interface for the XSLT engines. Public interface that extends IAltovaXMLEngine and IExecutable.

Superinterface: IAltovaXMLEngine, IExecutable, IReleasable
Implementing classes: XSLT1 and XSLT2

Note: The XSLT 2.0 Engine can be used in its backward compatibility mode to process an XSLT 1.0 stylesheet. The output, however, could be different than that produced by the XSLT 1.0 Engine processing the same XSLT 1.0 stylesheet.

Methods
The following methods are defined.

setXSLTFileName
public void **setXSLTFileName**(java.lang.String name)
Sets the file name for the XSLT data.
Parameters:
name: an absolute URL giving the base location of the XSLT data file.

setXSLTFromText
public void **setXSLTFromText**(java.lang.String text)
Sets text value for the XSLT data.
Parameters:
text: a string containing serialized XSLT data.

addExternalParameter
public void **addExternalParameter**(java.lang.String name,
java.lang.String val)
Adds the name and value of an external parameter.
Parameters:
name: a string containing a valid QName as the parameter name.
val: a string containing the value of the parameter; the value will be evaluated as an XPath expression.

clearExternalParameterList
public void **clearExternalParameterList**()
Clears the list of external parameters.

setXSLTStackSize

`public void` **`addExternalParameter`**`( long nVal)`
The stack size is the maximum depth of executed instructions. If the stack size is exceeded during a transformation, an error is reported.
Parameters:
`nVal`: numeric value for new stack size. Must be greater tha 100. The initial value 1000.

2.3.2 Classes

Given below is a summary of the classes of `com.altova.engines`. Detailed descriptions are given in the respective sections.

- `AltovaXMLFactory`
  Creates new AltovaXML COM server object instance via native call, and provides access to AltovaXML engines.
- `XMLValidator`
  Class holding XMLValidator.
- `XQuery`
  Class holding the XQuery 1.0 Engine.
- `XSLT1`
  Class holding the XSLT 1.0 Engine.
- `XSLT2`
  Class holding the XSLT 2.0 Engine.

AltovaXMLFactory

```
public class AltovaXMLFactory
extends java.lang.Object
implements IAltovaXMLFactory
```

Iplemented interfaces: `IAltovaXMLFactory`, `IReleasable`

Description
Creates new AltovaXML COM server object instance via native call, and provides access to the AltovaXML engines. The relationship between `AltovaXMLFactory` and the AltovaXML COM object is one-to-one. This means that subsequent calls to the `getENGINENAMEInstance()` function will return interfaces for the same engine instance.

Methods
The following methods are defined.

`getInstance`
`public static IAltovaXMLFactory` **`getInstance`**`()`
Creates a new `AltovaXMLFactory` object and connects it to a new AltovaXML COM server object.
Returns:
the interface `IAltovaXMLFactory` for the newly created `AltovaXMLFactory` object or null if the creation of the COM object failed. In the latter case you should make sure that `AltovaXML.exe` is properly registered as a COM server object.

`releaseInstance`
`public void` **`releaseInstance`**`()`
Releases the object's connection to the COM server.
Specified by:
`releaseInstance` in interface `IReleasable`.

`getXQueryInstance`
`public IXQuery` **`getXQueryInstance`**`()`

Creates a new instance of XQuery class for the current XQuery engine instance. The object's connection to the engine must be released after use. To do this, use the function releaseInstance() declared in the IReleasable interface.
Specified by:
getXQueryInstance in interface IAltovaXMLFactory.
Returns:
the IXQuery interface of the newly created class.

getXSLT1Instance
public IXSLT **getXSLT1Instance**()
Creates a new instance of XSLT1 class for the current XSLT 1.0 engine instance. The object's connection to the engine must be released after use. To do this, use the function releaseInstance() declared in the IReleasable interface.
Specified by:
getXSLT1Instance in interface IAltovaXMLFactory.
Returns:
the IXSLT interface of the newly created class.

getXSLT2Instance
public IXSLT **getXSLT2Instance**()
Creates a new instance of XSLT2 class for the current XSLT 2.0 engine instance. The object's connection to the engine must be released after use. To do this, use the function releaseInstance() declared in the IReleasable interface.
Specified by:
getXSLT2Instance in interface IAltovaXMLFactory.
Returns:
the IXSLT interface of the newly created class.

getXMLValidatorInstance
public IXMLValidator **getXMLValidatorInstance**()
Creates a new instance of XMLValidator class for the current XML Validator instance. The object's connection to the engine must be released after use. To do this, use the function releaseInstance() declared in the IReleasable interface.
Specified by:
getXMLValidatorInstance in interface IAltovaXMLFactory.
Returns:
the IXMLValidator interface of the newly created class.

XMLValidator

```
public class XMLValidator
extends java.lang.Object
implements IXMLValidator
```

Iplemented interfaces: IAltovaXMLEngine, IReleasable, IXMLValidator

Description
Class holding XMLValidator. No direct construction/access possible. Get the IXMLValidator interface to it by calling the function getXMLValidatorInstance() on an instance of IAltovaXMLFactory.

Constructors
The following constructor is defined.

XMLValidator
`protected` **XMLValidator**`( long nValidatorPtr)`

Methods
The following methods are defined.

releaseInstance
`public void` **releaseInstance**`( )`
Releases the object's connection to the COM server.
Specified by:
releaseInstance in interface IReleasable.

setInputXMLFileName
`public void` **setInputXMLFileName**`( java.lang.String str)`
Sets the file name for the input XML data. Note that you must use absolute URLs.
Specified by:
setInputXMLFileName in interface IAltovaXMLEngine.
Parameters:
`str`: an absolute URL giving the base location of the XML data.

setInputXMLFromText
`public void` **setInputXMLFromText**`( java.lang.String str)`
Sets the text value for the input XML data. Example: `setInputXMLFromText( "<doc> <a>text</a> </doc>" )`
Specified by:
setInputXMLFromText in interface IAltovaXMLEngine.
Parameters:
`str`: a string containing XML data.

getLastErrorMessage
`public java.lang.String` **getLastErrorMessage**`( )`
Gets the last error message from the engine.
Specified by:
getLastErrorMessage in interface IAltovaXMLEngine.
Returns:
a string containing the last error message.

isValid
`public boolean` **isValid**`( )`
Validates the input XML data against the DTD/Schema specified in it.
Specified by:
isValid in interface IXMLValidator.
Returns:
`true` on success, `false` on failure. In case of failure, you can use the function getLastErrorMessage declared in IAltovaXMLEngine to get additional information.

isWellFormed
`public boolean` **isWellFormed**`()`
Checks the input XML data for well-formedness.
Specified by:
`isWellFormed` in interface `IXMLValidator`.
Returns:
`true` on success, `false` on failure. In case of failure, you can use the function `getLastErrorMessage` declared in `IAltovaXMLEngine` to get additional information.

isValidWithExternalSchemaOrDTD
`public boolean` **isValidWithExternalSchemaOrDTD**`()`
Validates the input XML data against the external DTD/Schema, which can be specified with the functions `setDTDFileName()`, `setDTDFromText()`, `setSchemaFileName()`, and `setSchemaFromText()`. *For a description of these functions, see below*.
Specified by:
`isValidWithExternalSchemaOrDTD` in interface `IXMLValidator`.
Returns:
`true` on success, `false` on failure. In case of failure, you can use the function `getLastErrorMessage` declared in `IAltovaXMLEngine` to get additional information.

setSchemaFileName
`public void` **setSchemaFileName**`(java.lang.String str)`
Set file name of external Schema.
Specified by:
`setSchemaFileName` in interface `IXMLValidator`.
Parameters:
`str`: an absolute URL giving the base location of the Schema.

setDTDFileName
`public void` **setDTDFileName**`(java.lang.String str)`
Set file name of external DTD.
Specified by:
`setDTDFileName` in interface `IXMLValidator`.
Parameters:
`str`: an absolute URL giving the base location of the DTD.

setSchemaFromText
`public void` **setSchemaFromText**`(java.lang.String str)`
Sets text value for external Schema.
Specified by:
setSchemaFromText in interface `IXMLValidator`.
Parameters:
`str`: a string containing Schema as text.

setDTDFromText
`public void` **setDTDFromText**`(java.lang.String str)`
Sets text value for external DTD.

Specified by:
setDTDFromText in interface IXMLValidator.
Parameters:
str: a string containing DTD as text.

XQuery

```
public class XQuery
extends java.lang.Object
implements IXQuery
```

Iplemented interfaces: IAltovaXMLEngine, IExecutable, IReleasable, IXQuery

Description
Class holding the XQuery 1.0 engine. No direct construction/access possible. Get the IXQuery interface to it by calling the function getXQueryInstance() on an instance of IAltovaXMLFactory.

Constructors
The following constructor is defined.

XQuery
```
protected XQuery(long nXQueryPtr)
```

Methods
The following methods are defined.

releaseInstance
```
public void releaseInstance()
```
Releases the object's connection to the COM server.
Specified by:
releaseInstance in interface IReleasable.

execute
```
public boolean execute(java.lang.String sOutFile)
```
Executes and saves the result to file. In case of an error, you can use the function getLastErrorMessage() declared in IAltovaXMLEngine to get additional information.
Specified by:
execute in interface IExecutable.
Parameters:
sOutFile: an absolute URL giving the location of the output file.
Returns:
true on success, false on error.

executeAndGetResultAsString
```
public java.lang.String executeAndGetResultAsString()
```
Executes and returns the result as a UTF-16 text string. In case of an error, you can use the function getLastErrorMessage() declared in IAltovaXMLEngine to get additional information.

Specified by:
executeAndGetResultAsString in interface IExecutable.
Returns:
string containing the serialized result. On error, will return the empty string.

setInputXMLFileName
public void **setInputXMLFileName**(java.lang.String str)
Sets the file name for the input XML data. Note that you must use absolute URLs.
Specified by:
setInputXMLFileName in interface IAltovaXMLEngine.
Parameters:
str: an absolute URL giving the base location of the XML data.

setInputXMLFromText
public void **setInputXMLFromText**(java.lang.String str)
Sets the text value for the input XML data. Example: setInputXMLFromText("<doc><a>text </doc>").
Specified by:
setInputXMLFromText in interface IAltovaXMLEngine.
Parameters:
str: a string containing XML data.

getLastErrorMessage
public java.lang.String **getLastErrorMessage**( )
Gets the last error message from the engine.
Specified by:
getLastErrorMessage in interface IAltovaXMLEngine.
Returns:
a string containing the last error message.

setXQueryFileName
public void **setXQueryFileName**(java.lang.String str)
Sets file name of the XQuery document.
Specified by:
setXQueryFileName in interface IXQuery.
Parameters:
str: an absolute URL giving the base location of the XQuery file.

setXQueryStatement
public void **setXQueryStatement**(java.lang.String str)
Sets the text value for the XQuery statement.
Specified by:
setXQueryStatement in interface IXQuery
Parameters:
str: a string containing the XQuery statement.

setOutputEncoding
public void **setOutputEncoding**(java.lang.String str)

Sets the encoding for the result document.
Specified by:
setOutputEncoding in interface IXQuery.
Parameters:
str: a string containing an encoding name (for example: UTF-8, UTF-16, ASCII, 8859-1, 1252)

getOutputEncoding
public java.lang.String **getOutputEncoding**()
Retrieves the encoding specified for the result document.
Specified by:
getOutputEncoding in interface IXQuery.
Returns:
a string containing the encoding name.

setOutputIndent
public void **setOutputIndent**(boolean bVal)
Enables/disables the indentation option for the result document.
Specified by:
setOutputIndent in interface IXQuery.
Parameters:
bVal: boolean value to enable/disable indentation.

getOutputIndent
public boolean **getOutputIndent**()
Retrieves the output indent option specified for the result document.
Specified by:
getOutputIndent in interface IXQuery.
Returns:
the current value of the indent serialization parameter.

setOutputMethod
public void **setOutputMethod**(java.lang.String str)
Sets the serialization method for the result document.
Specified by:
setOutputMethod in interface IXQuery.
Parameters:
str: a string containing the serialization method. Valid values: xml, xhtml, html, text.

getOutputMethod
public java.lang.String **getOutputMethod**()
Retrieves the serialization method for the result document.
Specified by:
getOutputMethod in interface IXQuery.
Returns:
the current serialization method.

setOutputOmitXMLDeclaration

```
public void setOutputOmitXMLDeclaration( boolean bVal)
```
Enables/disables the serialization option `omitXMLDeclaration` for the result document.
Specified by:
`setOutputOmitXMLDeclaration` in interface `IXQuery`.
Parameters:
`bVal`: a new boolean value for the `omit-xml-declaration` parameter.

getOutputOmitXMLDeclaration

```
public boolean getOutputOmitXMLDeclaration( )
```
Retrieves the value of `omitXMLDeclaration` option specified for the result document.
Specified by:
`getOutputOmitXMLDeclaration` in interface `IXQuery`.
Returns:
boolean value of the `omit-xml-declaration` parameter.

addExternalVariable

```
public void addExternalVariable( java.lang.String strName,
                                 java.lang.String strVal)
```
Adds the name and value of an external variable.
Specified by:
`addExternalVariable` in interface `IXQuery`.
Parameters:
`strName`: a string containing a valid QName as the variable name.
`strVal`: a string containing the value of the variable; this value will be used as a string.

addExternalVariableAsXPath

```
public void addExternalVariableAsXPath( java.lang.String strName,
                                        java.lang.String strVal)
```
Add name and value for an external variable, with value being evaluated as an XPath 2.0 expression.
Specified by:
`addExternalVariableAsXPath` in interface `IXQuery`.
Parameters:
`strName`: a string containing a valid QName as the variable name.
`strVal`: a string containing the value of the variable; the value will be evaluated as an XPath 2.0 expression.

clearExternalVariableList

```
public void clearExternalVariableList( )
```
Clear the list of external variables.
Specified by:
`clearExternalVariableList` in interface `IXQuery`.

enableJavaExtensions

```
public void enableJavaExtensions( boolean bEnable)
```
Enable/disable Java extension functions.
Specified by:
`enableJavaExtensions` in interface `IExecutable`.

enableDotNetExtensions
`public void` **enableDotNetExtensions**`( boolean bEnable)`
Enable/disable .NET extension functions.
Specified by:
`enableJavaExtensions` in interface `IExecutable`.

XSLT1

```
public class XSLT1
extends java.lang.Object
implements IXSLT
```

Iplemented interfaces: `IAltovaXMLEngine`, `IExecutable`, `IReleasable`, `IXSLT`

Description
Class holding the XSLT 1.0 engine. No direct construction/access possible. Get the `IXSLT` interface to it by calling the function `getXSLT1Instance()` on an instance of `IAltovaXMLFactory`.

Constructors
The following constructor is defined.

XSLT1
`protected` **XSLT1**`( long nXSLT1Ptr)`

Methods
The following methods are defined.

releaseInstance
`public void` **releaseInstance**`( )`
Releases the object's connection to the COM server.
Specified by:
`releaseInstance` in interface `IReleasable`.

execute
`public boolean` **execute**`( java.lang.String sOutFile)`
Executes and saves the result to file. In case of an error, you can use the function `getLastErrorMessage()` declared in `IAltovaXMLEngine` to get additional information.
Specified by:
`execute` in interface `IExecutable`.
Parameters:
`sOutFile`: an absolute URL giving the location of the output file.
Returns:
`true` on success, `false` on error.

executeAndGetResultAsString
`public java.lang.String` **executeAndGetResultAsString**`( )`
Executes and returns the result as a UTF-16 text string. In case of an error, you can use the function `getLastErrorMessage()` declared in `IAltovaXMLEngine` to get additional information.

Specified by:
executeAndGetResultAsString in interface IExecutable.
Returns:
string containing the serialized result. On error, will return the empty string.

setInputXMLFileName
`public void setInputXMLFileName(java.lang.String str)`
Sets the file name for the input XML data. Note that you have to use absolute URLs.
Specified by:
setInputXMLFileName in interface IAltovaXMLEngine.
Parameters:
`str`: an absolute URL giving the base location of the XML data.

setInputXMLFromText
`public void setInputXMLFromText(java.lang.String str)`
Sets the text value for the input XML data. Example: `setInputXMLFromText( "<doc><a>text</a> </doc>" ).`
Specified by:
setInputXMLFromText in interface IAltovaXMLEngine.
Parameters:
`str`: a string containing XML data.

getLastErrorMessage
`public java.lang.String getLastErrorMessage()`
Gets the last error message from the engine.
Specified by:
getLastErrorMessage in interface IAltovaXMLEngine.
Returns:
a string containing the last error message.

setXSLTFileName
`public void setXSLTFileName(java.lang.String str)`
Sets the file name for the XSLT data.
Specified by:
setXSLTFileName in interface IXSLT.
Parameters:
`str`: an absolute URL giving the base location of the XSLT data

setXSLTFromText
`public void setXSLTFromText(java.lang.String str)`
Sets the text value for the XSLT data.
Specified by:
setXSLTFromText in interface IXSLT.
Parameters:
`str`: a string containing serialized XSLT data.

addExternalParameter
```
public void addExternalParameter(java.lang.String strName,
                                 java.lang.String strVal)
```


Adds the name and value of an external parameter.
Specified by:
addExternalParameter in interface IXSLT.
Parameters:
strName: a string containing a valid QName as the parameter name.
strVal: a string containing the value of the parameter; this value will be evaluated as an XPath expression.

clearExternalParameterList
public void **clearExternalParameterList**()
Clears the list of external parameters.
Specified by:
clearExternalParameterList in interface IXSLT.

setXSLTStackSize
public void **addExternalParameter**(long nVal)
The stack size is the maximum depth of executed instructions. If the stack size is exceeded during a transformation, an error is reported.
Specified by:
setXSLTStackSize in interface IXSLT.
Parameters:
nVal: numeric value for new stack size. Must be greater tha 100. The initial value 1000.

enableJavaExtensions
public void **enableJavaExtensions**(boolean bEnable)
Enable/disable Java extension functions.
Specified by:
enableJavaExtensions in interface IExecutable.

enableDotNetExtensions
public void **enableDotNetExtensions**(boolean bEnable)
Enable/disable .NET extension functions.
Specified by:
enableJavaExtensions in interface IExecutable.

XSLT2

```
public class XSLT2
extends java.lang.Object
implements IXSLT
```

Iplemented interfaces: IAltovaXMLEngine, IExecutable, IReleasable, IXSLT

Description
Class holding the XSLT 2.0 engine. No direct construction/access possible. Get the IXSLT interface to it by calling the function getXSLT2Instance() on an instance of IAltovaXMLFactory. Note that the XSLT 2.0 Engine can be used in its backward compatibility mode to process an XSLT 1.0 stylesheet. The output, however, could be different than that produced by the XSLT 1.0 Engine processing the same XSLT 1.0 stylesheet.

Constructors
The following constructor is defined.

XSLT2
`protected` **XSLT2**`( long nXSLT2Ptr)`

Methods
The following methods are defined.

releaseInstance
`public void` **releaseInstance**`()`
Releases the object's connection to the COM server.
Specified by:
`releaseInstance` in interface `IReleasable`.

execute
`public boolean` **execute**`( java.lang.String sOutFile)`
Executes and saves the result to file. In case of an error, you can use the function `getLastErrorMessage()` declared in `IAltovaXMLEngine` to get additional information.
Specified by:
`execute` in interface `IExecutable`.
Parameters:
`sOutFile`: an absolute URL giving the location of the output file.
Returns:
`true` on success, `false` on error.

executeAndGetResultAsString
`public java.lang.String` **executeAndGetResultAsString**`()`
Executes and returns the result as a UTF-16 text string. In case of an error, you can use the function `getLastErrorMessage()` declared in `IAltovaXMLEngine` to get additional information.
Specified by:
`executeAndGetResultAsString` in interface `IExecutable`.
Returns:
string containing the serialized result. On error, will return the empty string.

setInputXMLFileName
`public void` **setInputXMLFileName**`( java.lang.String str)`
Sets the file name for the input XML data. Note that you have to use absolute URLs.
Specified by:
`setInputXMLFileName` in interface `IAltovaXMLEngine`.
Parameters:
`str`: an absolute URL giving the base location of the XML data.

setInputXMLFromText
`public void` **setInputXMLFromText**`( java.lang.String str)`
Sets the text value for the input XML data. Example: `setInputXMLFromText( "<doc><a>text</a> </doc>" )`.

Specified by:
setInputXMLFromText in interface IAltovaXMLEngine.
Parameters:
str: a string containing XML data.

getLastErrorMessage
public java.lang.String **getLastErrorMessage**()
Gets the last error message from the engine.
Specified by:
getLastErrorMessage in interface IAltovaXMLEngine.
Returns:
a string containing the last error message.

setXSLTFileName
public void **setXSLTFileName**(java.lang.String str)
Sets the file name for the XSLT data.
Specified by:
setXSLTFileName in interface IXSLT.
Parameters:
str: an absolute URL giving the base location of the XSLT data

setXSLTFromText
public void **setXSLTFromText**(java.lang.String str)
Sets the text value for the XSLT data.
Specified by:
setXSLTFromText in interface IXSLT.
Parameters:
str: a string containing serialized XSLT data.

addExternalParameter
public void **addExternalParameter**(java.lang.String strName,
java.lang.String strVal)
Adds the name and value of an external parameter.
Specified by:
addExternalParameter in interface IXSLT.
Parameters:
strName: a string containing a valid QName as the parameter name.
strVal: a string containing the value of the parameter; this value will be evaluated as an XPath expression.

clearExternalParameterList
public void **clearExternalParameterList**()
Clears the list of external parameters.
Specified by:
clearExternalParameterList in interface IXSLT.

setInitialTemplateName
public void **setInitialTemplateName**(java.lang.String str)
Sets the initial template name for the transformation.

setInitialTemplateMode
`public void` **setInitialTemplateMode**`( java.lang.String str)`
Sets the initial template mode for the transformation.

setXSLTStackSize
`public void` **addExternalParameter**`( long nVal)`
The stack size is the maximum depth of executed instructions. If the stack size is exceeded during a transformation, an error is reported.
Specified by:
`setXSLTStackSize` in interface `IXSLT`.
Parameters:
`nVal`: numeric value for new stack size. Must be greater tha 100. The initial value 1000.

enableJavaExtensions
`public void` **enableJavaExtensions**`( boolean bEnable)`
Enable/disable Java extension functions.
Specified by:
`enableJavaExtensions` in interface `IExecutable`.

enableDotNetExtensions
`public void` **enableDotNetExtensions**`( boolean bEnable)`
Enable/disable .NET extension functions.
Specified by:
`enableJavaExtensions` in interface `IExecutable`.

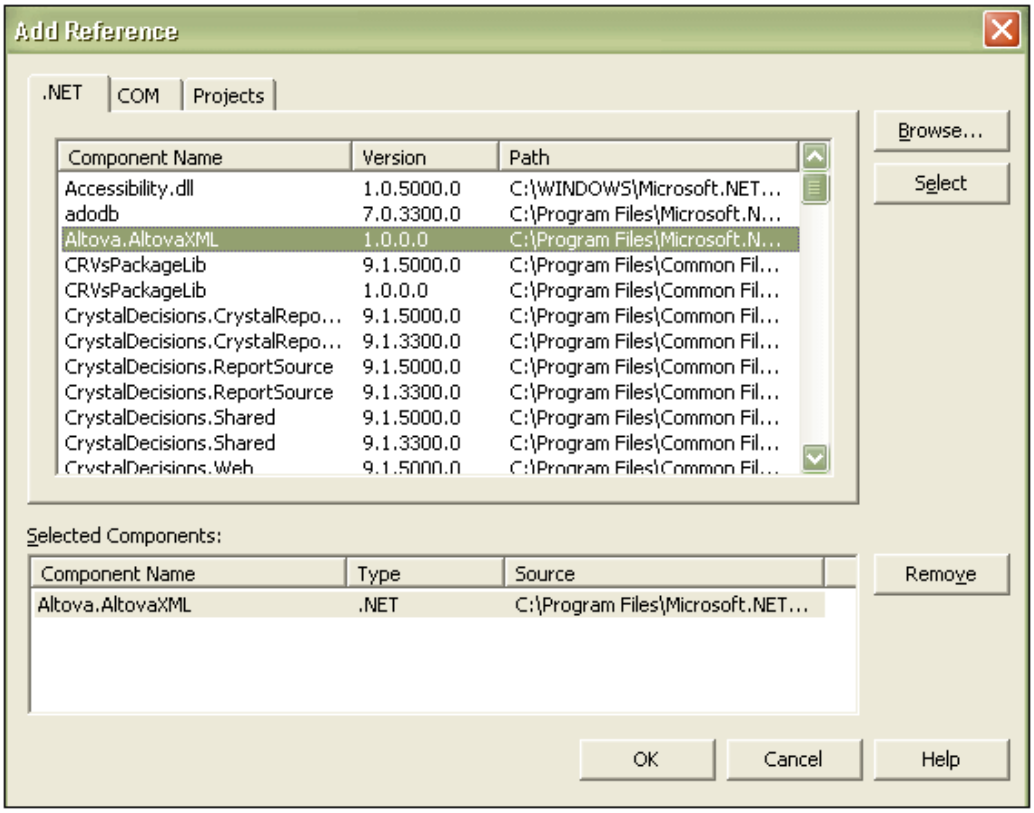
2.4 .NET Interface

The .NET interface is built as a wrapper around the AltovaXML COM interface. It is provided as a primary interop assembly signed by Altova and using the namespace `Altova.AltovaXML`. In order to use AltovaXML in your .NET project, you need to: (i) add a reference to the AltovaXML DLL (which is called `Altova.AltovaXML.dll`) in your project, and (ii) have AltovaXML registered as a COM server object. Once these requirements (which are described below) have been met, you can use the AltovaXML functionality in your project.

Adding the AltovaXML DLL as a reference to the project

The AltovaXML package contains a signed DLL file, named `Altova.AltovaXML.dll`, which will automatically be added to the global assembly cache (and the .NET reference library) when AltovaXML is installed using the AltovaXML installer. (It will be located typically in the `C:\WINDOWS\assembly` folder.) To add this DLL as a reference in a .NET project, do the following:

1. With the .NET project open, click **Project | Add Reference**. The Add Reference dialog (*screenshot below*) pops up, displaying a list of installed .NET components.



2. Select `Altova.AltovaXML` from the component list, double-click it or press the Select button, then click OK.

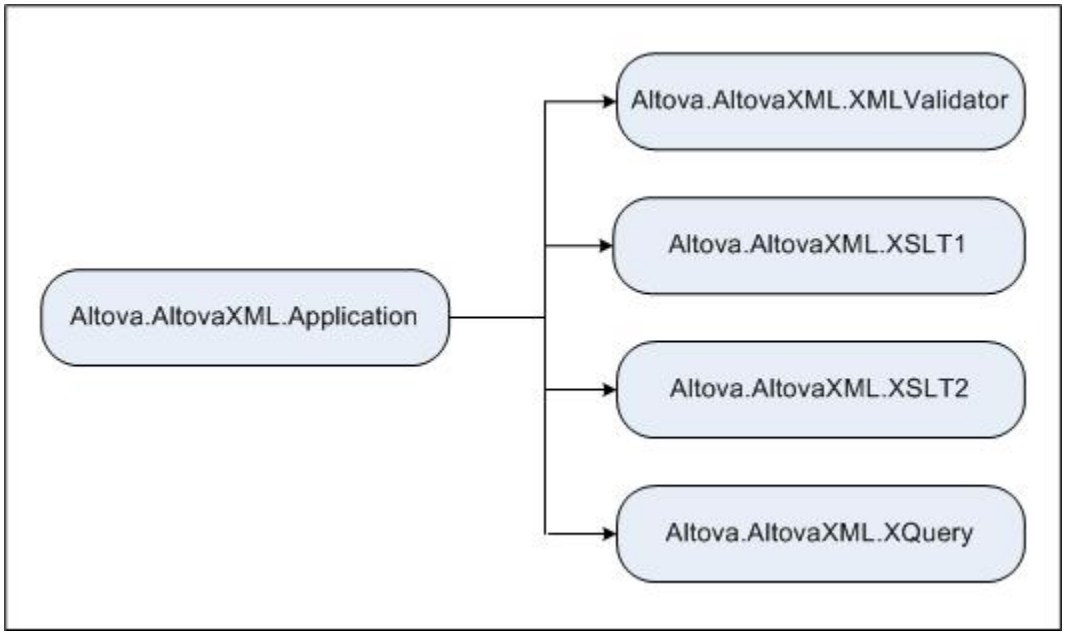
Registering AltovaXML as a COM server object

COM registration is done automatically by the AltovaXML Installer. If you change the location of the file `AltovaXML_COM.exe` after installation, you should register AltovaXML as a COM server object by running the command `AltovaXML_COM.exe /regserver`. (Note that the correct path to the `AltovaXML_COM.exe` must be entered. See Registering AltovaXML as a COM Server Object for more details.)

Once the `Altova.AltovaXML.dll` is available to the .NET interface and AltovaXML has been registered as a COM server object, AltovaXML functionality will be available in your .NET project.

2.4.1 General Usage and Example

The classes and methods you can use are as described in the COM Interface section, but are in the namespace `Altova.AltovaXML`. They are listed in the following sections. The starting point is the `Altova.AltovaXML.Application` object. When you create this object, a connection to a new AltovaXML COM server object is created. The object model is shown in the diagram below.



Example
How to use the AltovaXML classes and methods in the .NET framework is shown in the C# code for a button event listed below:

```
private void button1_Click(object sender, System.EventArgs e)
{
      Altova.AltovaXML.ApplicationClass appXML = new
Altova.AltovaXML.ApplicationClass();
      Altova.AltovaXML.XMLValidator XMLValidator =
appXML.XMLValidator;
      XMLValidator.InputXMLFromText = "<test>Is this data well-formed?
<a></test>";
      if ( XMLValidator.IsWellFormed() )
      {
      MessageBox.Show( this, "The input data is well-formed") ;
      }
      else
      {
      MessageBox.Show( this, "The input data is not well-formed") ;
      }
}
```

The code listing above does the following:

1. The `Altova.AltovaXML.ApplicationClass` object is created, which creates a connection to a new AltovaXML COM server object.
2. The XML Validator functionality is called using `Altova.AltovaXML.XMLValidator`.

3. The `InputXMLFromText` property of `Altova.AltovaXML.XMLValidator` submits the input XML data.
4. The `IsWellFormed` method of `Altova.AltovaXML.XMLValidator` checks whether the submitted XML data is well-formed, returning `TRUE` or `FALSE`.

For more detailed examples, see the example files in the `Examples` folder in the application folder.

2.4.2 Altova.AltovaXML.XMLValidator

Description
The `Altova.AltovaXML.XMLValidator` object provides methods to test:

- The well-formedness of an XML document.
- The validity of an XML document against a DTD or XML Schema referenced from within the XML document.
- The validity of an XML document against a DTD or XML Schema supplied externally via the code.
- The validity of an XBRL document against an XBRL taxonomy (a `.xsd` file).

All these methods return Boolean `TRUE` or `FALSE`.

Note: Where string inputs are to be interpreted as URLs, absolute paths should be used. If a relative path is used, a mechanism to resolve the relative path should be defined in the calling module.

Methods
The following methods are available:

`IsWellFormed`
`IsWellFormed` checks the well-formedness of the XML document. Returns `TRUE` if the XML document is well-formed, `FALSE` if it is not well-formed.

`IsValid`
`IsValid` validates the XML document against the DTD or XML Schema referenced in the XML document. Returns `TRUE` if the XML document is valid, `FALSE` if invalid. To validate against a DTD or XML Schema not referenced in the XML document, use the method `IsValidWithExternalSchemaOrDTD`.

`IsValidWithExternalSchemaOrDTD`
`IsValidWithExternalSchemaOrDTD` validates the XML document against the DTD or XML Schema supplied by any one of the following properties: `SchemaFileName`, `DTDFileName`, `SchemaFromText`, or `DTDFromText`. If more than one of these properties has values set for it, then the `IsValidWithExternalSchemaOrDTD` method uses the property that has been set last. Returns `TRUE` if the XML document is valid, `FALSE` if invalid. To validate against a DTD or XML Schema referenced in the XML document, use the method `IsValid`.

Note: Validation and well-formedness checks must always occur after assigning the XML and/or DTD or XML Schema document to the respective properties.

Properties
The following properties are defined:

`InputXMLFileName`
A string input that is read as a URL to locate the XML file to be validated.

`SchemaFileName`
A string input that is read as a URL to locate the XML Schema file against which the XML document is to be validated.

`DTDFileName`
A string input that is read as a URL to locate the DTD file against which the XML document is to be validated.

InputXMLFromText
A string input that constructs an XML document.

SchemaFromText
A string input that constructs an XML Schema document.

DTDFromText
A string input that constructs a DTD document.

LastErrorMessage
Returns the last error message.

2.4.3 Altova.AltovaXML.XSLT1

Description
The `Altova.AltovaXML.XSLT1` object provides methods and properties to execute an XSLT 1.0 transformation using the Altova XSLT 1.0 Engine. Results can be saved to a file or returned as a string. The object also enables XSLT parameters to be passed to the XSLT stylesheet. The URLs of XML and XSLT files can be supplied as strings via the object's properties. Alternatively, the XML and XSLT documents can be constructed within the code as text strings.

Note: Where string inputs are to be interpreted as URLs, absolute paths should be used. If a relative path is used, a mechanism to resolve the relative path should be defined in the calling module.

Methods
The following methods are available:

`Execute`
`Execute` executes an XSLT 1.0 transformation and saves the result to an output file, the name and location of which is provided as an input string to the `Execute` method.

`ExecuteAndGetResultAsString`
`ExecuteAndGetResultAsString` executes an XSLT 1.0 transformation and returns the result as a UTF-16 text string.

`AddExternalParameter`
Takes a parameter name and the value of this parameter as input arguments. Each external parameter and its value is to be specified in a separate call to the method. Providing an external parameter with the name of an existing (uncleared) parameter causes an error. Since parameter values are XPath expressions, parameter values that are strings must be enclosed in single quotes.

`ClearExternalParameterList`
No argument should be provided. The `ClearExternalParameterList` clears the external parameters list created with `AddExternalParameter` methods.

Note: Transformation must always occur after assigning the XML and XSLT documents.

Properties
The following properties are defined:

`InputXMLFileName`
A string input that is read as a URL to locate the XML file to be transformed.

`XSLFileName`
A string input that is read as a URL to locate the XSLT file to be used for the transformation.

`InputXMLFromText`
A string input that constructs an XML document.

`XSLFromText`
A string input that constructs an XSLT document.

`XSLStackSize`
The stack size is the maximum depth of executed instructions. The stack size can be changed with the `XSLStackSize` property. The minimum allowed stack size is 100. The default stack

size is 1000. If the stack size is exceeded during a transformation, an error is reported.

LastErrorMessage
Returns the last error message.

JavaExtensionsEnabled
Enables Java extensions. You can specify whether Java extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

DotNetExtensionsEnabled
Enables .NET extensions. You can specify whether .NET extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

2.4.4 Altova.AltovaXML.XSLT2

Description
The `Altova.AltovaXML.XSLT2` object provides methods and properties to execute an XSLT 2.0 transformation using the Altova XSLT 2.0 Engine. Results can be saved to a file or returned as a string. The object also enables XSLT parameters to be passed to the XSLT stylesheet. The URLs of XML and XSLT files can be supplied as strings via the object's properties. Alternatively, the XML and XSLT documents can be constructed within the code as text strings.

Note:

- Where string inputs are to be interpreted as URLs, absolute paths should be used. If a relative path is used, a mechanism to resolve the relative path should be defined in the calling module.
- The XSLT 2.0 Engine can be used in its backward compatibility mode to process an XSLT 1.0 stylesheet. The output, however, could be different than that produced by the XSLT 1.0 Engine processing the same XSLT 1.0 stylesheet.

Methods
The following methods are available:

`Execute`
`Execute` executes an XSLT 2.0 transformation and saves the result to an output file, the name and location of which is provided as an input string to the `Execute` method.

`ExecuteAndGetResultAsString`
`ExecuteAndGetResultAsString` executes an XSLT 2.0 transformation and returns the result as a UTF-16 text string.

`AddExternalParameter`
Takes a parameter name and the value of this parameter as input arguments. Each external parameter and its value is to be specified in a separate call to the method. Providing an external parameter with the name of an existing (uncleared) parameter causes an error. Since parameter values are XPath expressions, parameter values that are strings must be enclosed in single quotes.

`ClearExternalParameterList`
No argument should be provided. The `ClearExternalParameterList` clears the external parameters list created with `AddExternalParameter` methods.

`InitialTemplateName`
Sets the initial named template. The argument is the name of the template from which processing is to start.

`InitialTemplateMode`
Sets the initial mode for processing. The argument is the name of the required initial mode. Templates with this mode value will be processed.

Note: Transformation must always occur after assigning the XML and XSLT documents.

Properties
The following properties are defined:

`InputXMLFileName`
A string input that is read as a URL to locate the XML file to be transformed.

XSLFileName
A string input that is read as a URL to locate the XSLT file to be used for the transformation.

InputXMLFromText
A string input that constructs an XML document.

XSLFromText
A string input that constructs an XSLT document.

XSLStackSize
The stack size is the maximum depth of executed instructions. The stack size can be changed with the `XSLStackSize` property. The minimum allowed stack size is 100. The default stack size is 1000. If the stack size is exceeded during a transformation, an error is reported.

LastErrorMessage
Returns the last error message.

JavaExtensionsEnabled
Enables Java extensions. You can specify whether Java extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

DotNetExtensionsEnabled
Enables .NET extensions. You can specify whether .NET extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

2.4.5 Altova.AltovaXML.XQuery

Description
The `Altova.AltovaXML.XQuery` object provides methods and properties to execute an XQuery 1.0 transformation using the Altova XQuery 1.0 Engine. Results can be saved to a file or returned as a string. The object also enables external XQuery variables to be passed to the XQuery document. The URLs of XQuery and XML files can be supplied as strings via the object's properties. Alternatively, the XML and XQuery documents can be constructed within the code as text strings.

Note: Where string inputs are to be interpreted as URLs, absolute paths should be used. If a relative path is used, a mechanism to resolve the relative path should be defined in the calling module.

Methods
The following methods are available:

`Execute`
`Execute` executes an XQuery 1.0 transformation and saves the result to an output file, the name and location of which is provided as an input string to the `Execute` method.

`ExecuteAndGetResultAsString`
`ExecuteAndGetResultAsString` executes an XQuery 1.0 transformation and returns the result as a UTF-16 text string.

`AddExternalVariable`
Takes a variable name and the value of this variable as input arguments. Each external variable and its value is to be specified in a separate call to the method. Variables must be declared in the XQuery document, optionally with a type declaration. Whatever the type declaration for the external variable in the XQuery document, the variable value submitted to the `AddExternalVariable` does not need any special delimiter, such as quotes. However, the lexical form must match that of the expected type (for example, a variable of type `xs:date` must have a value in the lexical form `2004-01-31`; a value in the lexical form `2004/Jan/01` will cause an error). Note that this also means that you cannot use an XQuery 1.0 function (for example, `current-date()`) as the value of an external variable (since the lexical form of the function as it is written will either not match the required data type (if the datatype is specified in the declaration of the external variable) or will be read as a string (if the datatype is not specified).) Providing an external variable that has the name of an existing (uncleared) variable causes an error.

`ClearExternalVariableList`
No argument should be provided. The `ClearExternalVariableList` clears the external variables list created with `AddExternalVariable` methods.

Note: Setting the optional XML document must always be done before query execution.

Properties
The following properties are defined:

`XQueryFileName`
A string input that is read as a URL to locate the XQuery file to be executed. If both the `XQueryFileName` property and `XQueryFromText` property are specified, then the property that has been set later than the other (in the code sequence) is used.

InputXMLFileName
A string input that is read as a URL to locate the XML file that will be loaded into the query. XQuery navigation expressions are evaluated with reference to the document node of this XML document. If both the `InputXMLFileName` property and `InputXMLFromText` property are specified, then the property that has been set later than the other (in the code sequence) is used.

XQueryFromText
A string input that constructs an XQuery document. If both the `XQueryFileName` property and `XQueryFromText` property are specified, then the property that has been set later than the other (in the code sequence) is used.

InputXMLFromText
A string input that constructs an XML document. XQuery navigation expressions are evaluated with reference to the document node of this XML document. If both the `InputXMLFileName` property and `InputXMLFromText` property are specified, then the property that has been set later than the other (in the code sequence) is used.

LastErrorMessage
Returns the last error message.

JavaExtensionsEnabled
Enables Java extensions. You can specify whether Java extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

DotNetExtensionsEnabled
Enables .NET extensions. You can specify whether .NET extensions should be enabled or not by submitting `true` or `false` (case-insensitive) as a Boolean argument.

Note: If an XML document is set and is not needed for a new XQuery execution, then it should be cleared with an empty string assignment.

The following serialization options are defined:

OutputMethod
The required output method can be specified by submitting the required value as a string argument. Valid values are: `xml`, `xhtml`, `html`, and `text`. For example: `objAltovaXML.XQuery.OutputMethod = "xml"`. If the value is invalid, it is ignored. The default output method is `xml`.

OutputOmitXMLDeclaration
You can specify whether the XML declaration should be omitted or included in the output by submitting `true` or `false` (case-insensitive) as a Boolean argument. For example: `objAltovaXML.XQuery.OutputOmitXMLDeclaration = "FALSE"`. If the value is invalid, an error is raised. The default option is `TRUE`.

OutputIndent
You can specify whether the output should be indented or not by submitting `true` or `false` (case-insensitive) as a Boolean argument. For example: `objAltovaXML.XQuery.OutputIndent = "TRUE"`. If the value is invalid, an error is raised. The default option is False.

OutputEncoding
The required output encoding can be specified by submitting the encoding value as a string argument. For example: `objAltovaXML.XQuery.OutputEncoding = "UTF-8"`. If the value is invalid, it is ignored. The default output encoding is UTF-8.

Note: For the serialization options, Raw Interface and Dispatch Interface usage differs. In the Raw Interface, if no argument is provided with these properties, then the current value of the property is returned. You would use something like: `put_OutputOption( VARIANT_BOOL bVal )` or `VARIANT_BOOL bVal = get_OutputOption()`, respectively, to set values and get values. In the Dispatch Interface, you can use `b = myXQuery.OutputOption` to get values and `myXQuery.OutputOption = b` to set values. For example, in the Dispatch Interface, `Sheet1.Cells(10, 2) = objAltovaXML.XQuery.OutputEncoding` would get the current output encoding.

2.5 Explicitly releasing AltovaXML COM-Server from C# and VB.NET

It is possible to explicitly release the AltovaXML COM references from within C# code using the ReleaseComObject methods shown below.

Example:

```
        private void button1_Click(object sender, EventArgs e)
        {
            Altova.AltovaXML.ApplicationClass AltovaXML = new Altova.
AltovaXML.ApplicationClass();
            Altova.AltovaXML.IXSLT2 XSLT2 = AltovaXML.XSLT2;

            XSLT2.InputXMLFileName =
"C:\\Projects\\files\\XMLSpyExeFolder\\Examples\\OrgChart.xml";
            XSLT2.XSLFileName =
"C:\\Projects\\files\\XMLSpyExeFolder\\Examples\\OrgChart.xsl";
            XSLT2.Execute(
"C:\\Projects\\files\\XMLSpyExeFolder\\Examples\\OrgChart_out.html");

            // you must release ALL references to all components that
you received.
            System.Runtime.InteropServices.Marshal.ReleaseComObject(
XSLT2);
            XSLT2 = null;
            System.Runtime.InteropServices.Marshal.ReleaseComObject(
AltovaXML);
            AltovaXML = null;
        }
```

- At the end of the method, the AltovaXML.exe **server** shuts down!
- If you do not call **all** of the ReleaseComObject methods, the exe servers will only be shut down with the shutdown of the C# application.

2.6 OOXML and ZIP Files

In order to enforce output to a ZIP file, including Open Office XML (OOXML) files such as .docx, one must specify the ZIP protocol in the file path. For example:

```
filename.zip|zip/filename.xxx
filename.docx|zip/filename.xxx
```

In AltovaXML, ZIP file output can be specified with the following operations:

COM interface and .NET interface
Output is generated using the `Execute` method. The argument of the method specifies the output file's name and location. For ZIP files, the ZIP protocol must be used, as in the following examples:

```
xslt2.Execute(c:\Mydocs\orgchart.zip|zip\main.xml)
xslt2.Execute(c:\Mydocs\orgchart.docx|zip\main.out)
xslt2.Execute(c:\Mydocs\orgchart.docx|zip\)
```

Command line
When using the command line ensure that the output URI is enclosed in quotes. This is because the pipe character (|) would otherwise be interpreted by the command system. An example:

```
AltovaXML -in input.xml -xslt2 transform.xslt -out "c:\results.zipart.zip|
zip\result.xml"
```

The `xsl:result-document` element
In the case of the `xsl:result-document` element of XSLT 2.0, the ZIP protocol must be used on the output URI. In the case of OOXML documents, the ZIP protocol must be specified on the output URI of every `xsl:result-document` element involved in creating files for the OOXML document.

If the `xsl:result-document` elements specify relative output URIs, then specify the ZIP protocol for the main result, the URI of which is then used as the base URI to resolve the relative output URIs.

Chapter 3

Engine Information

3 Engine Information

This section contains information about implementation-specific features of the Altova XML Validator, Altova XSLT 1.0 Engine, Altova XSLT 2.0 Engine, and Altova XQuery Engine.

3.1 Altova XML Validator

The Altova XML Validator implements and conforms to the rules of:

- XML 1.0 (Fourth Edition)
- XML Namespaces (1.0)
- XML Schemas (Structures)
- XML Schema (Datatypes)

3.2 XSLT 1.0 Engine: Implementation Information

The Altova XSLT 1.0 Engine is built into Altova's XMLSpy, StyleVision, Authentic, and MapForce XML products. It is also available in the free AltovaXML package. The Altova XSLT 1.0 Engine implements and conforms to the World Wide Web Consortium's XSLT 1.0 Recommendation of 16 November 1999 and XPath 1.0 Recommendation of 16 November 1999 . Limitations and implementation-specific behavior are listed below.

Limitations

- The **xsl:preserve-space** and **xsl:strip-space** elements are not supported.
- When the `method` attribute of `xsl:output` is set to HTML, or if HTML output is selected by default, then special characters in the XML or XSLT file are inserted in the HTML document directly as special characters; they are not inserted as HTML character references in the output. For instance, the character ` ` (the decimal character reference for a non-breaking space) is not inserted as ` ` in the HTML code, but directly as a non-breaking space.

Implementation's handling of whitespace-only nodes in source XML document

The XML data (and, consequently, the XML Infoset) that is passed to the Altova XSLT 1.0 Engine is stripped of boundary-whitespace-only text nodes. (A boundary-whitespace-only text node is a child whitespace-only text node that occurs between two elements within an element of mixed content.) This stripping may have an effect on the value returned by the `fn:position()`, `fn:last()`, and `fn:count()` functions.

For any node selection that selects text nodes also, boundary-whitespace-only text nodes would typically also be included in the selection. However, since the XML Infoset used by the Altova engines has boundary-whitespace-only text nodes stripped from it, these nodes are not present in the XML Infoset. As a result, the size of the selection and the numbering of nodes in the selection will be different than that for a selection which included these text nodes. The `fn:position()`, `fn:last()`, and `fn:count()` functions, therefore, could produce results that are different from those produced by some other processors.

A situation in which boundary-whitespace-only text nodes are evaluated as siblings of other elements arises most commonly when `xsl:apply-templates` is used to apply templates. When the `fn:position()`, `fn:last()`, and `fn:count()` functions are used in patterns with a name test (for example, `para[3]`, which is short for `para[position()=3]`), boundary-whitespace-only nodes are irrelevant since only the named elements (`para` in the above example) are selected. (Note, however, that boundary-whitespace-only nodes **are** relevant in patterns that use the wildcard, for example, `*[10]`.)

Note: If a boundary-whitespace-only text node is required in the output, then insert the required whitespace within one of the two adjoining child elements. For example, the XML fragment:

```
<para>This is <b>bold</b> <i>italic</>.</para>
```

when processed with the XSLT template

```
<xsl:template match="para">
   <xsl:apply-templates/>
</xsl:template>
```

will produce:

```
This is bolditalic.
```


To get a space between `bold` and `italic` in the output, insert a space character within either the `<b>` or `<i>` elements in the XML source. For example:

```
<para>This is <b>bold</b> <i> italic</>.</para> or
<para>This is <b>bold&#x20;</b> <i>italic</>.</para> or
<para>This is <b>bold</b><i>&#x20;italic</>.</para>
```

When such an XML fragment is processed with the same XSLT template given above, it will produce:

```
This is bold italic.
```

3.3 XSLT 2.0 Engine: Implementation Information

The Altova XSLT 2.0 Engine is built into Altova's XMLSpy, StyleVision, Authentic, and MapForce XML products. It is also available in the free AltovaXML package. This section describes the engine's implementation-specific aspects of behavior. It starts with a section giving general information about the engine, and then goes on to list the implementation-specific behavior of XSLT 2.0 functions.

For information about implementation-specific behavior of XPath 2.0 functions, see the section, XPath 2.0 and XQuery 1.0 Functions.

3.3.1 General Information

The Altova XSLT 2.0 Engine conforms to the World Wide Web Consortium's (W3C's) XSLT 2.0 Recommendation of 23 January 2007. Note the following general information about the engine.

Backwards Compatibility
The Altova XSLT 2.0 Engine is backwards compatible. The only time the backwards compatibility of the XSLT 2.0 Engine comes into play is when using the XSLT 2.0 Engine of Altova XML to process an XSLT 1.0 stylesheet. Note that there could be differences in the outputs produced by the XSLT 1.0 Engine and the backwards-compatible XSLT 2.0 Engine.

In all other Altova products, the backwards-compatibility issue never arises. This is because these products automatically select the appropriate engine for the transformation. For example, consider that in XMLSpy you specify that a certain XML document be processed with an XSLT 1.0 stylesheet. When the transformation command is invoked, XMLSpy automatically selects the XSLT 1.0 Engine of XMLSpy to carry out the transformation.

Note: The stylesheet version is specified in the `version` attribute of the `stylesheet` or `transform` element of the stylesheet.

Namespaces
Your XSLT 2.0 stylesheet should declare the following namespaces in order for you to be able to use the type constructors and functions available in XSLT 2.0. The prefixes given below are conventionally used; you could use alternative prefixes if you wish.

Namespace Name	Prefix	Namespace URI
XML Schema types	`xs:`	`http://www.w3.org/2001/XMLSchema`
XPath 2.0 functions	`fn:`	`http://www.w3.org/2005/xpath-functions`

Typically, these namespaces will be declared on the `xsl:stylesheet` or `xsl:transform` element, as shown in the following listing:

```
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:fn="http://www.w3.org/2005/xpath-functions"
   ...
</xsl:stylesheet>
```

The following points should be noted:

- The Altova XSLT 2.0 Engine uses the XPath 2.0 and XQuery 1.0 Functions namespace (listed in the table above) as its **default functions namespace**. So you can use XPath 2.0 and XSLT 2.0 functions in your stylesheet without any prefix. If you declare the XPath 2.0 Functions namespace in your stylesheet with a prefix, then you can additionally use the prefix assigned in the declaration.
- When using type constructors and types from the XML Schema namespace, the prefix used in the namespace declaration must be used when calling the type constructor (for example, `xs:date`).
- With the CRs of 23 January 2007, the `untypedAtomic` and duration datatypes (`dayTimeDuration` and `yearMonthDuration`), which were formerly in the XPath Datatypes namespace (typically prefixed `xdt:`) have been moved to the XML Schema namespace.

- Some XPath 2.0 functions have the same name as XML Schema datatypes. For example, for the XPath functions `fn:string` and `fn:boolean` there exist XML Schema datatypes with the same local names: `xs:string` and `xs:boolean`. So if you were to use the XPath expression `string('Hello')`, the expression evaluates as `fn:string('Hello')` —not as `xs:string('Hello')`.

Schema-awareness
The Altova XSLT 2.0 Engine is schema-aware.

Whitespace in XML document
By default, the Altova XSLT 2.0 Engine strips all boundary whitespace from boundary-whitespace-only nodes in the source XML document. The removal of this whitespace affects the values that the `fn:position()`, `fn:last()`, `fn:count()`, and `fn:deep-equal()` functions return. For more details, see Whitespace-only Nodes in XML Document in the XPath 2.0 and XQuery 1.0 Functions section.

Note: If a boundary-whitespace-only text node is required in the output, then insert the required whitespace within one of the two adjoining child elements. For example, the XML fragment:

```
<para>This is <b>bold</b> <i>italic</>.</para>
```

when processed with the XSLT template

```
<xsl:template match="para">
   <xsl:apply-templates/>
</xsl:template>
```

will produce:

```
This is bolditalic.
```

To get a space between `bold` and `italic` in the output, insert a space character within either the `<b>` or `<i>` elements in the XML source. For example:

```
<para>This is <b>bold</b> <i> italic</>.</para> or
<para>This is <b>bold&#x20;</b> <i>italic</>.</para> or
<para>This is <b>bold</b><i>&#x20;italic</>.</para>
```

When such an XML fragment is processed with the same XSLT template given above, it will produce:

```
This is bold italic.
```

XSLT 2.0 elements and functions
Limitations and implementation-specific behavior of XSLT 2.0 elements and functions are listed in the section XSLT 2.0 Elements and Functions.

XPath 2.0 functions
Implementation-specific behavior of XPath 2.0 functions is listed in the section XPath 2.0 and XQuery 1.0 Functions.

3.3.2 XSLT 2.0 Elements and Functions

Limitations
The **xsl: preserve-space** and **xsl: strip-space** elements are not supported.

Implementation-specific behavior
Given below is a description of how the Altova XSLT 2.0 Engine handles implementation-specific aspects of the behavior of certain XSLT 2.0 functions.

xsl: result-document
Additionally supported encodings are: `base16tobinary` and `base64tobinary`.

function-available
The function tests for the availability of in-scope functions (XSLT 2.0, XPath 2.0, and extension functions).

unparsed-text
The `href` attribute accepts (i) relative paths for files in the base-uri folder, and (ii) absolute paths with or without the `file: //` protocol. Additionally supported encodings are: `binarytobase16` and `binarytobase64`.

3.4 XQuery 1.0 Engine: Implementation Information

The Altova XQuery 1.0 Engine is built into Altova's XMLSpy and MapForce XML products. It is also available in the free AltovaXML package. This section provides information about implementation-defined aspects of behavior.

Standards conformance
The Altova XQuery 1.0 Engine conforms to the World Wide Web Consortium's (W3C's) XQuery 1.0 Recommendation of 23 January 2007. The XQuery standard gives implementations discretion about how to implement many features. Given below is a list explaining how the Altova XQuery 1.0 Engine implements these features.

Schema awareness
The Altova XQuery 1.0 Engine is **schema-aware**.

Encoding
The UTF-8 and UTF-16 character encodings are supported.

Namespaces
The following namespace URIs and their associated bindings are pre-defined.

Namespace Name	Prefix	Namespace URI
XML Schema types	`xs:`	`http://www.w3.org/2001/XMLSchema`
Schema instance	`xsi:`	`http://www.w3.org/2001/XMLSchema-instance`
Built-in functions	`fn:`	`http://www.w3.org/2005/xpath-functions`
Local functions	`local:`	`http://www.w3.org/2005/xquery-local-functions`

The following points should be noted:

- The Altova XQuery 1.0 Engine recognizes the prefixes listed above as being bound to the corresponding namespaces.
- Since the built-in functions namespace listed above is the default functions namespace in XQuery, the `fn:` prefix does not need to be used when built-in functions are invoked (for example, `string("Hello")` will call the `fn:string` function). However, the prefix `fn:` can be used to call a built-in function without having to declare the namespace in the query prolog (for example: `fn:string("Hello")`).
- You can change the default functions namespace by declaring the `default function namespace` expression in the query prolog.
- When using types from the XML Schema namespace, the prefix `xs:` may be used without having to explicitly declare the namespaces and bind these prefixes to them in the query prolog. (Example: `xs:date` and `xs:yearMonthDuration`.) If you wish to use some other prefix for the XML Schema namespace, this must be explicitly declared in the query prolog. (Example: `declare namespace alt = "http://www.w3.org/2001/XMLSchema"; alt:date("2004-10-04")`.)
- Note that the `untypedAtomic`, `dayTimeDuration`, and `yearMonthDuration` datatypes have been moved, with the CRs of 23 January 2007, from the XPath Datatypes namespace to the XML Schema namespace, so: `xs:yearMonthDuration`.

If namespaces for functions, type constructors, node tests, etc are wrongly assigned, an error is

reported. Note, however, that some functions have the same name as schema datatypes, e.g. `fn:string` and `fn:boolean`. (Both `xs:string` and `xs:boolean` are defined.) The namespace prefix determines whether the function or type constructor is used.

XML source document and validation
XML documents used in executing an XQuery document with the Altova XQuery 1.0 Engine must be well-formed. However, they do not need to be valid according to an XML Schema. If the file is not valid, the invalid file is loaded without schema information. If the XML file is associated with an external schema and is valid according to it, then post-schema validation information is generated for the XML data and will be used for query evaluation.

Static and dynamic type checking
The static analysis phase checks aspects of the query such as syntax, whether external references (e.g. for modules) exist, whether invoked functions and variables are defined, and so on. No type checking is done in the static analysis phase. If an error is detected in the static analysis phase, it is reported and the execution is stopped.

Dynamic type checking is carried out at run-time, when the query is actually executed. If a type is incompatible with the requirement of an operation, an error is reported. For example, the expression `xs:string("1") + 1` returns an error because the addition operation cannot be carried out on an operand of type `xs:string`.

Library Modules
Library modules store functions and variables so they can be reused. The Altova XQuery 1.0 Engine supports modules that are stored in **a single external XQuery file**. Such a module file must contain a `module` declaration in its prolog, which associates a target namespace. Here is an example module:

```
module namespace libns="urn:module-library";
declare variable $libns:company := "Altova";
declare function libns: webaddress() { "http://www.altova.com" };
```

All functions and variables declared in the module belong to the namespace associated with the module. The module is used by importing it into an XQuery file with the `import module` statement in the query prolog. The `import module` statement only imports functions and variables declared directly in the library module file. As follows:

```
import module namespace modlib = "urn:module-library" at
   "modulefilename.xq";
if    ($modlib:company = "Altova")
then   modlib:webaddress()
else   error("No match found.")
```

External functions
External functions are not supported, i.e. in those expressions using the `external` keyword, as in:

```
declare function hoo($param as xs:integer) as xs:string external;
```

Collations
The default collation is the Unicode codepoint collation. No other collation is currently supported. Comparisons, including the `fn:max` function, are based on this collation.

Character normalization
No character normalization form is supported.

Precision of numeric types

- The `xs:integer` datatype is arbitrary-precision, i.e. it can represent any number of digits.
- The `xs:decimal` datatype has a limit of 20 digits after the decimal point.
- The `xs:float` and `xs:double` datatypes have limited-precision of 15 digits.

XQuery Instructions Support
The `Pragma` instruction is not supported. If encountered, it is ignored and the fallback expression is evaluated.

XQuery Functions Support
For information about implementation-specific behavior of XQuery 1.0 functions, see the section, XPath 2.0 and XQuery 1.0 Functions.

3.5 XPath 2.0 and XQuery 1.0 Functions

XPath 2.0 and XQuery 1.0 functions are evaluated by:

- the **Altova XPath 2.0 Engine**, which (i) is a component of the Altova XSLT 2.0 Engine, and (ii) is used in the XPath Evaluator of Altova's XMLSpy product to evaluate XPath expressions with respect to the XML document that is active in the XMLSpy interface.
- the **Altova XQuery 1.0 Engine**.

This section describes how XPath 2.0 and XQuery 1.0 functions are handled by the Altova XPath 2.0 Engine and Altova XQuery 1.0 Engine. Only those functions are listed, for which the behavior is implementation-specific, or where the behavior of an individual function is different in any of the three environments in which these functions are used (that is, in XSLT 2.0, in XQuery 1.0, and in the XPath Evaluator of XMLSpy). Note that this section does not describe how to use these functions. For more information about the usage of functions, see the World Wide Web Consortium's (W3C's) XQuery 1.0 and XPath 2.0 Functions and Operators Recommendation of 23 January 2007.

3.5.1 General Information

Standards conformance

- The Altova XPath 2.0 Engine implements the World Wide Web Consortium's (W3C's) XPath 2.0 Recommendation of 23 January 2007. The Altova XQuery 1.0 Engine implements the World Wide Web Consortium's (W3C's) XQuery 1.0 Recommendation of 23 January 2007. The XPath 2.0 and XQuery 1.0 functions support in these two engines is compliant with the XQuery 1.0 and XPath 2.0 Functions and Operators Recommendation of 23 January 2007.
- The Altova XPath 2.0 Engine conforms to the rules of XML 1.0 (Fourth Edition) and XML Namespaces (1.0).

Default functions namespace
The default functions namespace has been set to comply with that specified in the standard. Functions can therefore be called without a prefix.

Boundary-whitespace-only nodes in source XML document
The XML data (and, consequently, the XML Infoset) that is passed to the Altova XPath 2.0 Engine and Altova XQuery 1.0 Engine is stripped of boundary-whitespace-only text nodes. (A boundary-whitespace-only text node is a child whitespace-only text node that occurs between two elements within an element of mixed content.) This stripping has an effect on the value returned by the `fn:position()`, `fn:last()`, `fn:count()`, and `fn:deep-equal()` functions.

For any node selection that selects text nodes also, boundary-whitespace-only text nodes would typically also be included in the selection. However, since the XML Infoset used by the Altova engines has boundary-whitespace-only text nodes stripped from it, these nodes are not present in the XML Infoset. As a result, the size of the selection and the numbering of nodes in the selection will be different than that for a selection which included these text nodes. The `fn:position()`, `fn:last()`, `fn:count()`, and `fn:deep-equal()` functions, therefore, could produce results that are different from those produced by some other processors.

A situation in which boundary-whitespace-only text nodes are evaluated as siblings of other elements arises most commonly when `xsl:apply-templates` is used to apply templates. When the `fn:position()`, `fn:last()`, and `fn:count()` functions are used in patterns with a name test (for example, `para[3]`, which is short for `para[position()=3]`), boundary-whitespace-only nodes are irrelevant since only the named elements (`para` in the above example) are selected. (Note, however, that boundary-whitespace-only nodes **are** relevant in patterns that use the wildcard, for example, `*[10]`.)

Numeric notation
On output, when an `xs:double` is converted to a string, scientific notation (for example, `1.0E12`) is used when the absolute value is less than 0.000001 or greater than 1,000,000. Otherwise decimal or integer notation is used.

Precision of `xs:decimal`
The precision refers to the number of digits in the number, and a minimum of 18 digits is required by the specification. For division operations that produce a result of type `xs:decimal`, the precision is 19 digits after the decimal point with no rounding.

Implicit timezone

When two `date`, `time`, or `dateTime` values need to be compared, the timezone of the values being compared need to be known. When the timezone is not explicitly given in such a value, the implicit timezone is used. The implicit timezone is taken from the system clock, and its value can be checked with the `fn:implicit-timezone()` function.

Collations
Only the Unicode codepoint collation is supported. No other collations can be used. String comparisons, including for the `fn:max` and `fn:min` functions, are based on this collation.

Namespace axis
The namespace axis is deprecated in XPath 2.0. Use of the namespace axis is, however, supported. To access namespace information with XPath 2.0 mechanisms, use the `fn:in-scope-prefixes()`, `fn:namespace-uri()` and `fn:namespace-uri-for-prefix()` functions.

Static typing extensions
The optional static type checking feature is not supported.

3.5.2 Functions Support

The table below lists (in alphabetical order) the implementation-specific behavior of certain functions. The following general points should be noted:

- In general, when a function expects a sequence of one item as an argument, and a sequence of more than one item is submitted, then an error is returned.
- All string comparisons are done using the Unicode codepoint collation.
- Results that are QNames are serialized in the form `[ prefix: ] localname`.

Function Name	Notes
`base-uri`	• If external entities are used in the source XML document and if a node in the external entity is specified as the input node argument of the `base-uri()` function, it is still the base URI of the including XML document that is used—not the base URI of the external entity. • The base URI of a node in the XML document can be modified using the `xml:base` attribute.

`collection`	• The argument is a relative URI that is resolved against the current base URI.
	• If the resolved URI identifies an XML file, then this XML file is treated as a catalog which references a collection of files. This file must have the form:
	`<collection>`
	`  <doc href="uri-1" />`
	`  <doc href="uri-2" />`
	`  <doc href="uri-3" />`
	`</collection>`
	The files referenced by the `href` attributes are loaded, and their document nodes are returned as a sequence.
	• If the resolved URI does not identify an XML file with the catalog structure described above, then the argument string (in which wildcards such as `?` and `*` are allowed) is used as a search string. XML files with names that match the search expression are loaded, and their document nodes are returned as a sequence. See examples below.
	• XSLT example: The expression `collection("c:\MyDocs\*.xml")//Title` returns a sequence of all `DocTitle` elements in the `.xml` files in the `MyDocs` folder.
	• XQuery example: The expression `{for $i in collection(c:\MyDocs\*.xml) return element doc{base-uri($i)}}` returns the base URIs of all the `.xml` files in the `MyDocs` folder, each URI being within a `doc` element.
	• The default collection is empty.

Function Name	Notes
`count`	• See note on whitespace in the General Information section.

`current-date`, `current-dateTime`, `current-time`	• The current date and time is taken from the system clock. • The timezone is taken from the implicit timezone provided by the evaluation context; the implicit timezone is taken from the system clock. • The timezone is always specified in the result.
`deep-equal`	• See note on whitespace in the General Information section.
`doc`	• An error is raised only if no XML file is available at the specified location or if the file is not well-formed. The file is validated if a schema is available. If the file is not valid, the invalid file is loaded without schema information.
`id`	• In a well-formed but invalid document that contains two or more elements having the same ID value, the first element in document order is returned.
`in-scope-prefixes`	• Only default namespaces may be undeclared in the XML document. However, even when a default namespace is undeclared on an element node, the prefix for the default namespace, which is the zero-length string, is returned for that node.
`last`	• See note on whitespace in the General Information section.
`lower-case`	• The Unicode character set is supported.
`normalize-unicode`	• The normalization forms NFC, NFD, NFKC, and NFKD are supported.

Function Name	Notes
`position`	• See note on whitespace in the General Information section.
`resolve-uri`	• If the second, optional argument is omitted, the URI to be resolved (the first argument) is resolved against the base URI from the static context, which is the URI of the XSLT stylesheet or the base URI given in the prolog of the XQuery document. • The relative URI (the first argument) is appended after the last `"/"` in the path notation of the base URI notation. • If the value of the first argument is the zero-length string, the base URI from the static context is returned, and this URI includes the file name of the document from which the base URI of the static context is derived (e.g. the XSLT or XML file).

static-base-uri	• The base URI from the static context is the base URI of the XSLT stylesheet or the base URI specified in the prolog of the XQuery document. • When using XPath Evaluator in the XMLSpy IDE, the base URI from the static context is the URI of the active XML document.
upper-case	• The Unicode character set is supported.

3.6 Extensions

There are several ready-made functions in programming languages such as Java and C# that are not available as XPath 2.0 / XQuery 1.0 functions or as XSLT 2.0 functions. A good example of such functions are the math functions available in Java, such as `sin()` and `cos()`. If these functions were available to the designers of XSLT stylesheets and XQuery queries, it would increase the application area of stylesheets and queries and greatly simplify the tasks of stylesheet creators.

Altova Engines (XSLT 1.0, XSLT 2.0, and XQuery 1.0), which are used in a number of Altova products, support the use of extension functions in Java and .NET. The Altova XSLT Engines additionally support MSXSL scripts for XSLT 1.0 and 2.0 and Altova's own extension functions.

You should note that extension functions are always called from XPath expressions. This section describes how to use extension functions and MSXSL scripts in your XSLT stylesheets and XQuery queries. These descriptions are organized into the following sections:

- Java Extension Functions
- .NET Extension Functions
- MSXSL Scripts for XSLT
- Altova Extension Functions

The two main issues considered in the descriptions are: (i) how functions in the respective libraries are called; and (ii) what rules are followed for converting arguments in a function call to the required input format of the function, and what rules are followed for the return conversion (function result to XSLT/XQuery data object).

Requirements
For extension functions support, a Java Runtime Environment (for access to Java functions) and .NET Framework 2.0 (minimum, for access to .NET functions) must be installed on the machine running the XSLT transformation or XQuery execution, or must be accessible for the transformations.

3.6.1 Java Extension Functions

A Java extension function can be used within an XPath or XQuery expression to invoke a Java constructor or call a Java method (static or instance).

A field in a Java class is considered to be a method without any argument. A field can be static or instance. How to access fields is described in the respective sub-sections, static and instance.

This section is organized into the following sub-sections:

- Java: Constructors
- Java: Static Methods and Static Fields
- Java: Instance Methods and Instance Fields
- Datatypes: XSLT/XQuery to Java
- Datatypes: Java to XSLT/XQuery

Form of the extension function
The extension function in the XPath/XQuery expression must have the form `prefix:fname()`.

- The `prefix:` part identifies the extension function as a Java function. It does so by associating the extension function with an in-scope namespace declaration, the URI of which must begin with `java:` (*see below for examples*). The namespace declaration should identify a Java class, for example: `xmlns:myns="java:java.lang.Math"`. However, it could also simply be: `xmlns:myns="java"` (without a colon), with the identification of the Java class being left to the `fname()` part of the extension function.
- The `fname()` part identifies the Java method being called, and supplies the arguments for the method (*see below for examples*). However, if the namespace URI identified by the `prefix:` part does not identify a Java class (*see preceding point*), then the Java class should be identified in the `fname()` part, before the class and separated from the class by a period (*see the second XSLT example below*).

Note: The class being called must be on the classpath of the machine.

XSLT example
Here are two examples of how a static method can be called. In the first example, the class name (`java.lang.Math`) is included in the namespace URI and, therefore, must not be in the `fname()` part. In the second example, the `prefix:` part supplies the prefix `java:` while the `fname()` part identifies the class as well as the method.

```
<xsl:value-of xmlns:jMath="java:java.lang.Math"
              select="jMath:cos(3.14)" />

<xsl:value-of xmlns:jmath="java"
              select="jmath:java.lang.Math.cos(3.14)" />
```

The method named in the extension function (`cos()` in the example above) must match the name of a public static method in the named Java class (`java.lang.Math` in the example above).

XQuery example
Here is an XQuery example similar to the XSLT example above:

```
<cosine xmlns:jMath="java:java.lang.Math">
   {jMath:cos(3.14)}
</cosine>
```

User-defined Java class files
If you have created your own Java classes, methods in these classes are called differently according to: (i) whether the classes are accessed via a JAR file or a class file, and (ii) whether these files (JAR or class) are located in the current directory (the same directory as the XSLT or XQuery document) or not. Since the built-in Java classes and Java classes in the current directory are found when a Java function is executed, there is no need to specify the location of class files in the current directory. However, paths to class files not in the current directory and to all JAR files must be specified.

Class files
If access is via a class file, then there are two possibilities:

- The class file is in the same folder as the XSLT or XQuery document. In this case, since all classes in the folder are found, the file location does not need to be specified. The syntax to identify a class is:

  `java:classname`

  where

  `java`: indicates that a user-defined Java function is being called; (Java classes in the current directory will be loaded by default)
  `classname` is the name of the required method's class

  The class is identified in a namespace URI, and the namespace is used to prefix a method call.

  The example below calls the `getVehicleType()` method of the `Car` class of the `com.altova.extfunc` package:

```
<xsl:stylesheet version="2.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:xs="http://www.w3.org/2001/XMLSchema"
       xmlns:fn="http://www.w3.org/2005/xpath-functions"
       xmlns:car="java:com.altova.extfunc.Car" >
<xsl:output exclude-result-prefixes="fn car xsl fo xs"/>

<xsl:template match="/">
   <a>
   <xsl:value-of select="car:getVehicleType()"/>
   </a>
</xsl:template>

</xsl:stylesheet>
```

- The class file is not in the same folder as the XSLT or XQuery document. In this case, the location of the class file must be specified within the URI as a query string. The syntax is:

  `java:classname[?path=uri-of-classfile]`

  where

  `java`: indicates that a user-defined Java function is being called
  `uri-of-classfile` is the URI of the the class file

`classname` is the name of the required method's class

The class is identified in a namespace URI, and the namespace is used to prefix a method call.

The example below shows how to access a class file that is located in another directory than the current directory.

```
<xsl:stylesheet version="2.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:fn="http://www.w3.org/2005/xpath-functions"
      xmlns:car="
java:Car?path=file:C:/test/classExample/com.altova.extfunc" >

<xsl:output exclude-result-prefixes="fn car xsl xs"/>

<xsl:template match="/">
   <xsl:variable name="myCar" select="car:new('red')" />
   <a><xsl:value-of select="car:getCarColor($myCar)"/></a>
</xsl:template>

</xsl:stylesheet>
```

Note: When a path is supplied via the extension function, the path is added to the ClasssLoader.

JAR files

If access is via a JAR file, the URI of the JAR file must be specified using the following syntax:

```
xmlns:classNS="java:classname?path=jar:uri-of-jarfile!/"
```

The method is then called by using the prefix of the namespace URI that identifies the class: `classNS:method()`

In the above:

`java:` indicates that a Java function is being called
`classname` is the name of the user-defined class
`?` is the separator between the classname and the path
`path=jar:` indicates that a path to a JAR file is being given
`uri-of-jarfile` is the URI of the jar file
`!/` is the end delimiter of the path
`classNS:method()` is the call to the method

Alternatively, the classname can be given with the method call. Here are two examples of the syntax:

```
xmlns:ns1="java:docx.layout.pages?path=jar:file://c:/projects/docs/docx.jar!/"
      ns1:main()

      xmlns:ns2="java?path=jar:file://c:/projects/docs/docx.jar!/"
      ns2:docx.layout.pages.main()
```

Here is a complete XSLT example that uses a JAR file to call a Java extension function:

```
<xsl:stylesheet version="2.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:fn="http://www.w3.org/2005/xpath-functions"
      xmlns:car="java?path=jar:file://C:/test/Car1.jar!/" >
<xsl:output exclude-result-prefixes="fn car xsl xs"/>

<xsl:template match="/">
      <xsl:variable name="myCar" select="car:Car1.new('red')" />
      <a><xsl:value-of select="car:Car1.getCarColor($myCar)"/></a>
</xsl:template>

<xsl:template match="car"/>

</xsl:stylesheet>
```

Note: When a path is supplied via the extension function, the path is added to the ClassLoader.

Java: Constructors

An extension function can be used to call a Java constructor. All constructors are called with the pseudo-function `new()`.

If the result of a Java constructor call can be implicitly converted to XPath/XQuery datatypes, then the Java extension function will return a sequence that is an XPath/XQuery datatype. If the result of a Java constructor call cannot be converted to a suitable XPath/XQuery datatype, then the constructor creates a wrapped Java object with a type that is the name of the class returning that Java object. For example, if a constructor for the class `java.util.Date` is called (`java.util.Date.new()`), then an object having a type `java.util.Date` is returned. The lexical format of the returned object may not match the lexical format of an XPath datatype and the value would therefore need to be converted to the lexical format of the required XPath datatype and then to the required XPath datatype.

There are two things that can be done with a Java object created by a constructor:

- It can be assigned to a variable:
  ```
  <xsl:variable name="currentdate" select="date:new()" xmlns:date="
  java:java.util.Date" />
  ```
- It can be passed to an extension function (*see Instance Method and Instance Fields*):
  ```
  <xsl:value-of select="date:toString(date:new())" xmlns:date="
  java:java.util.Date" />
  ```

Java: Static Methods and Static Fields

A static method is called directly by its Java name and by supplying the arguments for the method. Static fields (methods that take no arguments), such as the constant-value fields `E` and `PI`, are accessed without specifying any argument.

XSLT examples
Here are some examples of how static methods and fields can be called:

```
<xsl:value-of xmlns:jMath="java:java.lang.Math"
              select="jMath:cos(3.14)" />

<xsl:value-of xmlns:jMath="java:java.lang.Math"
              select="jMath:cos( jMath:PI() )" />
```

```
<xsl:value-of xmlns:jMath="java:java.lang.Math"
              select="jMath:E() * jMath:cos(3.14)" />
```

Notice that the extension functions above have the form `prefix:fname()`. The prefix in all three cases is `jMath:`, which is associated with the namespace URI `java:java.lang.Math`. (The namespace URI must begin with `java:`. In the examples above it is extended to contain the class name (`java.lang.Math`).) The `fname()` part of the extension functions must match the name of a public class (e.g. `java.lang.Math`) followed by the name of a public static method with its argument/s (such as `cos(3.14)`) or a public static field (such as `PI()`).

In the examples above, the class name has been included in the namespace URI. If it were not contained in the namespace URI, then it would have to be included in the `fname()` part of the extension function. For example:

```
<xsl:value-of xmlns:java="java:"
              select="java:java.lang.Math.cos(3.14)" />
```

XQuery example
A similar example in XQuery would be:

```
<cosine xmlns:jMath="java:java.lang.Math">
   {jMath:cos(3.14)}
</cosine>
```

Java: Instance Methods and Instance Fields

An instance method has a Java object passed to it as the first argument of the method call. Such a Java object typically would be created by using an extension function (for example a constructor call) or a stylesheet parameter/variable. An XSLT example of this kind would be:

```
<xsl:stylesheet version="1.0" exclude-result-prefixes="date"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:date="java:java.util.Date"
  xmlns:jlang="java:java.lang">
  <xsl:param name="CurrentDate" select="date:new()"/>
  <xsl:template match="/">
      <enrollment institution-id="Altova School"
                  date="{date:toString($CurrentDate)}"
                  type="{jlang:Object.toString(jlang:Object.getClass(  date:new()
))}">
      </enrollment>
  </xsl:template>
</xsl:stylesheet>
```

In the example above, the value of the node `enrollment/@type` is created as follows:

1. An object is created with a constructor for the class `java.util.Date` (with the `date:new()` constructor).
2. This Java object is passed as the argument of the `jlang.Object.getClass` method.
3. The object obtained by the `getClass` method is passed as the argument to the `jlang.Object.toString` method.

The result (the value of `@type`) will be a string having the value: `java.util.Date`.

An instance field is theoretically different from an instance method in that it is not a Java object per se that is passed as an argument to the instance field. Instead, a parameter or variable is passed as the argument. However, the parameter/variable may itself contain the value returned by a Java object. For example, the parameter `CurrentDate` takes the value returned by a constructor for the class `java.util.Date`. This value is then passed as an argument to the

instance method `date:toString` in order to supply the value of `/enrollment/@date`.

Datatypes: XPath/XQuery to Java

When a Java function is called from within an XPath/XQuery expression, the datatype of the function's arguments is important in determining which of multiple Java classes having the same name is called.

In Java, the following rules are followed:

- If there is more than one Java method with the same name, but each has a different number of arguments than the other/s, then the Java method that best matches the number of arguments in the function call is selected.
- The XPath/XQuery string, number, and boolean datatypes (*see list below*) are implicitly converted to a corresponding Java datatype. If the supplied XPath/XQuery type can be converted to more than one Java type (for example, `xs:integer`), then that Java type is selected which is declared for the selected method. For example, if the Java method being called is `fx(decimal)` and the supplied XPath/XQuery datatype is `xs:integer`, then `xs:integer` will be converted to Java's `decimal` datatype.

The table below lists the implicit conversions of XPath/XQuery string, number, and boolean types to Java datatypes.

`xs:string`	`java.lang.String`
`xs:boolean`	`boolean` (primitive), `java.lang.Boolean`
`xs:integer`	int, long, short, byte, float, double, and the wrapper classes of these, such as `java.lang.Integer`
`xs:float`	`float` (primitive), `java.lang.Float`, `double` (primitive)
`xs:double`	`double` (primitive), `java.lang.Double`
`xs:decimal`	`float` (primitive), `java.lang.Float`, `double` (primitive), `java.lang.Double`

Subtypes of the XML Schema datatypes listed above (and which are used in XPath and XQuery) will also be converted to the Java type/s corresponding to that subtype's ancestor type.

In some cases, it might not be possible to select the correct Java method based on the supplied information. For example, consider the following case.

- The supplied argument is an `xs:untypedAtomic` value of 10 and it is intended for the method `mymethod(float)`.
- However, there is another method in the class which takes an argument of another datatype: `mymethod(double)`.
- Since the method names are the same and the supplied type (`xs:untypedAtomic`) could be converted correctly to either `float` or `double`, it is possible that `xs:untypedAtomic` is converted to `double` instead of `float`.
- Consequently the method selected will not be the required method and might not produce the expected result. To work around this, you can create a user-defined method with a different name and use this method.

Types that are not covered in the list above (for example `xs:date`) will not be converted and will

generate an error. However, note that in some cases, it might be possible to create the required Java type by using a Java constructor.

Datatypes: Java to XPath/XQuery

When a Java method returns a value, the datatype of the value is a string, numeric or boolean type, then it is converted to the corresponding XPath/XQuery type. For example, Java's `java.lang.Boolean` and `boolean` datatypes are converted to `xsd:boolean`.

One-dimensional arrays returned by functions are expanded to a sequence. Multi-dimensional arrays will not be converted, and should therefore be wrapped.

When a wrapped Java object or a datatype other than string, numeric or boolean is returned, you can ensure conversion to the required XPath/XQuery type by first using a Java method (e.g `toString`) to convert the Java object to a string. In XPath/XQuery, the string can be modified to fit the lexical representation of the required type and then converted to the required type (for example, by using the `cast as` expression).

3.6.2 .NET Extension Functions

If you are working on the .NET platform, you can use extension functions written in any of the .NET languages (for example, C#). A .NET extension function can be used within an XPath or XQuery expression to invoke a constructor, property, or method (static or instance) within a .NET class.

A property of a .NET class is called using the syntax `get_PropertyName()`.

This section is organized into the following sub-sections:

- .NET: Constructors
- .NET: Static Methods and Static Fields
- .NET: Instance Methods and Instance Fields
- Datatypes: XSLT/XQuery to .NET
- Datatypes: .NET to XSLT/XQuery

Form of the extension function
The extension function in the XPath/XQuery expression must have the form `prefix:fname()`.

- The `prefix:` part is associated with a URI that identifies the .NET class being addressed.
- The `fname()` part identifies the constructor, property, or method (static or instance) within the .NET class, and supplies any argument/s, if required.
- The URI must begin with `clitype:` (which identifies the function as being a .NET extension function).
- The `prefix:fname()` form of the extension function can be used with system classes and with classes in a loaded assembly. However, if a class needs to be loaded, additional parameters containing the required information will have to be supplied.

Parameters
To load an assembly, the following parameters are used:

`asm`	The name of the assembly to be loaded.
`ver`	The version number (maximum of four integers separated by periods).
`sn`	The key token of the assembly's strong name (16 hex digits).
`from`	A URI that gives the location of the assembly (DLL) to be loaded. If the URI is relative, it is relative to the XSLT or XQuery document. If this parameter is present, any other parameter is ignored.
`partialname`	The partial name of the assembly. It is supplied to `Assembly.LoadWith.PartialName()`, which will attempt to load the assembly. If `partialname` is present, any other parameter is ignored.
`loc`	The locale, for example, `en-US`. The default is `neutral`.

If the assembly is to be loaded from a DLL, use the `from` parameter and omit the `sn` parameter. If the assembly is to be loaded from the Global Assembly Cache (GAC), use the `sn` parameter and omit the `from` parameter.

A question mark must be inserted before the first parameter, and parameters must be separated by a semi-colon. The parameter name gives its value with an equals sign (*see example below*).

Examples of namespace declarations

An example of a namespace declaration in XSLT that identifies the system class `System.Environment`:

```
xmlns:myns="clitype:System.Environment"
```

An example of a namespace declaration in XSLT that identifies the class to be loaded as `Trade.Forward.Scrip`:

```
xmlns:myns="clitype:Trade.Forward.Scrip?asm=forward;version=10.6.2.1"
```

An example of a namespace declaration in XQuery that identifies the system class `MyManagedDLL.testClass`:. Two cases are distinguished:

1. When the assembly is loaded from the GAC:

```
declare namespace cs="clitype:MyManagedDLL.testClass?asm=MyManagedDLL;
ver=1.2.3.4;loc=neutral;sn=b9f091b72dccfba8";
```

2. When the assembly is loaded from the DLL (complete and partial references below):

```
declare namespace
cs="clitype:MyManagedDLL.testClass?from=file:///C:/Altova
Projects/extFunctions/MyManagedDLL.dll;

declare namespace
cs="clitype:MyManagedDLL.testClass?from=MyManagedDLL.dll;
```

XSLT example

Here is a complete XSLT example that calls functions in system class `System.Math`:

```
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:fn="http://www.w3.org/2005/xpath-functions">
   <xsl:output method="xml" omit-xml-declaration="yes" />
   <xsl:template match="/">
      <math xmlns:math="clitype:System.Math">
         <sqrt><xsl:value-of select="math:Sqrt(9)"/></sqrt>
         <pi><xsl:value-of select="math:PI()"/></pi>
         <e><xsl:value-of select="math:E()"/></e>
         <pow><xsl:value-of select="math:Pow(math:PI(), math:E())"/></pow>
      </math>
   </xsl:template>
</xsl:stylesheet>
```

The namespace declaration on the element `math` associates the prefix `math:` with the URI `clitype:System.Math`. The `clitype:` beginning of the URI indicates that what follows identifies either a system class or a loaded class. The `math:` prefix in the XPath expressions associates the extension functions with the URI (and, by extension, the class) `System.Math`. The extension functions identify methods in the class `System.Math` and supply arguments where required.

XQuery example

Here is an XQuery example fragment similar to the XSLT example above:

```
<math xmlns:math="clitype:System.Math">
   {math:Sqrt(9)}
</math>
```

As with the XSLT example above, the namespace declaration identifies the .NET class, in this

case a system class. The XQuery expression identifies the method to be called and supplies the argument.

.NET: Constructors

An extension function can be used to call a .NET constructor. All constructors are called with the pseudo-function `new()`. If there is more than one constructor for a class, then the constructor that most closely matches the number of arguments supplied is selected. If no constructor is deemed to match the supplied argument/s, then a `'No constructor found'` error is returned.

Constructors that return XPath/XQuery datatypes

If the result of a .NET constructor call can be implicitly converted to XPath/XQuery datatypes, then the .NET extension function will return a sequence that is an XPath/XQuery datatype.

Constructors that return .NET objects

If the result of a .NET constructor call cannot be converted to a suitable XPath/XQuery datatype, then the constructor creates a wrapped .NET object with a type that is the name of the class returning that object. For example, if a constructor for the class `System.DateTime` is called (with `System.DateTime.new()`), then an object having a type `System.DateTime` is returned.

The lexical format of the returned object may not match the lexical format of a required XPath datatype. In such cases, the returned value would need to be: (i) converted to the lexical format of the required XPath datatype; and (ii) cast to the required XPath datatype.

There are three things that can be done with a .NET object created by a constructor:

- It can be used within a variable:
```
<xsl:variable name="currentdate" select="date:new(2008, 4, 29)"
xmlns:date="clitype:System.DateTime" />
```
- It can be passed to an extension function (*see Instance Method and Instance Fields*):
```
<xsl:value-of select="date:ToString(date:new(2008, 4, 29))" xmlns:date
="clitype:System.DateTime" />
```
- It can be converted to a string, number, or boolean:
-
```
<xsl:value-of select="xs:integer(data:get_Month(date:new(2008, 4, 29)))
" xmlns:date="clitype:System.DateTime" />
```

.NET: Static Methods and Static Fields

A static method is called directly by its name and by supplying the arguments for the method. The name used in the call must exactly match a public static method in the class specified. If the method name and the number of arguments that were given in the function call matches more than one method in a class, then the types of the supplied arguments are evaluated for the best match. If a match cannot be found unambiguously, an error is reported.

Note: A field in a .NET class is considered to be a method without any argument. A property is called using the syntax `get_PropertyName()`.

Examples

An XSLT example showing a call to a method with one argument (`System.Math.Sin(arg)`):

```
<xsl:value-of select="math:Sin(30)" xmlns:math="clitype:System.Math"/>
```

An XSLT example showing a call to a field (considered a method with no argument) (`System.Double.MaxValue()`):

```
<xsl:value-of select="double:MaxValue()" xmlns:double="
clitype:System.Double"/>
```

An XSLT example showing a call to a property (syntax is `get_PropertyName()`) (`System.String()`):

```
<xsl:value-of select="string:get_Length('my string')" xmlns:string="
clitype:System.String"/>
```

An XQuery example showing a call to a method with one argument (`System.Math.Sin(arg)`):

```
<sin xmlns:math="clitype:System.Math">
   { math:Sin(30) }
</sin>
```

.NET: Instance Methods and Instance Fields

An instance method has a .NET object passed to it as the first argument of the method call. This .NET object typically would be created by using an extension function (for example a constructor call) or a stylesheet parameter/variable. An XSLT example of this kind would be:

```
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:fn="http://www.w3.org/2005/xpath-functions">
   <xsl:output method="xml" omit-xml-declaration="yes"/>
   <xsl:template match="/">
      <xsl:variable name="releasedate"
         select="date:new(2008, 4, 29)"
         xmlns:date="clitype:System.DateTime"/>
      <doc>
         <date>
            <xsl:value-of select="date:ToString(date:new(2008, 4, 29))"
               xmlns:date="clitype:System.DateTime"/>
         </date>
         <date>
            <xsl:value-of select="date:ToString($releasedate)"
               xmlns:date="clitype:System.DateTime"/>
         </date>
      </doc>
   </xsl:template>
</xsl:stylesheet>
```

In the example above, a `System.DateTime` constructor (`new(2008, 4, 29)`) is used to create a .NET object of type `System.DateTime`. This object is created twice, once as the value of the variable `releasedate`, a second time as the first and only argument of the `System.DateTime.ToString()` method. The instance method `System.DateTime.ToString()` is called twice, both times with the `System.DateTime` constructor (`new(2008, 4, 29)`) as its first and only argument. In one of these instances, the variable `releasedate` is used to get the .NET object.

Instance methods and instance fields

The difference between an instance method and an instance field is theoretical. In an instance method, a .NET object is directly passed as an argument; in an instance field, a parameter or variable is passed instead—though the parameter or variable may itself contain a .NET object. For example, in the example above, the variable `releasedate` contains a .NET object, and it is this variable that is passed as the argument of `ToString()` in the second `date` element constructor. Therefore, the `ToString()` instance in the first `date` element is an instance method while the second is considered to be an instance field. The result produced in both instances, however, is the same.

Datatypes: XPath/XQuery to .NET

When a .NET extension function is used within an XPath/XQuery expression, the datatypes of the function's arguments are important for determining which one of multiple .NET methods having the same name is called.

In .NET, the following rules are followed:

- If there is more than one method with the same name in a class, then the methods available for selection are reduced to those that have the same number of arguments as the function call.
- The XPath/XQuery string, number, and boolean datatypes (*see list below*) are implicitly converted to a corresponding .NET datatype. If the supplied XPath/XQuery type can be converted to more than one .NET type (for example, `xs:integer`), then that .NET type is selected which is declared for the selected method. For example, if the .NET method being called is `fx(double)` and the supplied XPath/XQuery datatype is `xs:integer`, then `xs:integer` will be converted to .NET's `double` datatype.

The table below lists the implicit conversions of XPath/XQuery string, number, and boolean types to .NET datatypes.

`xs:string`	`StringValue, string`
`xs:boolean`	`BooleanValue, bool`
`xs:integer`	`IntegerValue, decimal, long, integer, short, byte, double, float`
`xs:float`	`FloatValue, float, double`
`xs:double`	`DoubleValue, double`
`xs:decimal`	`DecimalValue, decimal, double, float`

Subtypes of the XML Schema datatypes listed above (and which are used in XPath and XQuery) will also be converted to the .NET type/s corresponding to that subtype's ancestor type.

In some cases, it might not be possible to select the correct .NET method based on the supplied information. For example, consider the following case.

- The supplied argument is an `xs:untypedAtomic` value of 10 and it is intended for the method `mymethod(float)`.
- However, there is another method in the class which takes an argument of another datatype: `mymethod(double)`.
- Since the method names are the same and the supplied type (`xs:untypedAtomic`) could be converted correctly to either `float` or `double`, it is possible that `xs:untypedAtomic` is converted to `double` instead of `float`.
- Consequently the method selected will not be the required method and might not produce the expected result. To work around this, you can create a user-defined

method with a different name and use this method.

Types that are not covered in the list above (for example `xs:date`) will not be converted and will generate an error.

Datatypes: .NET to XPath/XQuery

When a .NET method returns a value and the datatype of the value is a string, numeric or boolean type, then it is converted to the corresponding XPath/XQuery type. For example, .NET's `decimal` datatype is converted to `xsd:decimal`.

When a .NET object or a datatype other than string, numeric or boolean is returned, you can ensure conversion to the required XPath/XQuery type by first using a .NET method (for example `System.DateTime.ToString()`) to convert the .NET object to a string. In XPath/XQuery, the string can be modified to fit the lexical representation of the required type and then converted to the required type (for example, by using the `cast as` expression).

3.6.3 MSXSL Scripts for XSLT

The <msxsl:script> element contains user-defined functions and variables that can be called from within XPath expressions in the XSLT stylesheet. The <msxsl:script> is a top-level element, that is, it must be a child element of <xsl:stylesheet> or <xsl:transform>.

The <msxsl:script> element must be in the namespace urn:schemas-microsoft-com:xslt (*see example below*).

Scripting language and namespace

The scripting language used within the block is specified in the <msxsl:script> element's language attribute and the namespace to be used for function calls from XPath expressions is identified with the implements-prefix attribute (*see below*).

```
<msxsl:script language="scripting-language" implements-prefix="user-namespace-
prefix">

   function-1 or variable-1
   ...
   function-n or variable-n

</msxsl:script>
```

The <msxsl:script> element interacts with the Windows Scripting Runtime, so only languages that are installed on your machine may be used within the <msxsl:script> element. **The .NET Framework 2.0 platform or higher must be installed for MSXSL scripts to be used.** Consequently, the .NET scripting languages can be used within the <msxsl:script> element.

The language attribute accepts the same values as the language attribute on the HTML <script> element. If the language attribute is not specified, then Microsoft JScript is assumed as the default.

The implements-prefix attribute takes a value that is a prefix of a declared in-scope namespace. This namespace typically will be a user namespace that has been reserved for a function library. All functions and variables defined within the <msxsl:script> element will be in the namespace identified by the prefix specified in the implements-prefix attribute. When a function is called from within an XPath expression, the fully qualified function name must be in the same namespace as the function definition.

Example

Here is an example of a complete XSLT stylesheet that uses a function defined within a <msxsl:script> element.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:fn="http://www.w3.org/2005/xpath-functions"
   xmlns:msxsl="urn:schemas-microsoft-com:xslt"
   xmlns:user="http://mycompany.com/mynamespace">

  <msxsl:script language="VBScript" implements-prefix="user">
    <![CDATA[
    ' Input: A currency value: the wholesale price
    ' Returns: The retail price: the input value plus 20% margin,
    ' rounded to the nearest cent
    dim a as integer  = 13
    Function AddMargin(WholesalePrice) as integer
```

```
      AddMargin = WholesalePrice * 1.2 + a
    End Function
  ]]>
  </msxsl:script>

  <xsl:template match="/">
    <html>
      <body>
        <p>
          <b>Total Retail Price =
            $<xsl:value-of select="user:AddMargin(50)"/>
          </b>
          <br/>
          <b>Total Wholesale Price =
            $<xsl:value-of select="50"/>
          </b>
        </p>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Datatypes

The values of parameters passed into and out of the script block are limited to XPath datatypes. This restriction does not apply to data passed among functions and variables within the script block.

Assemblies

An assembly can be imported into the script by using the **msxsl:assembly** element. The assembly is identified via a name or a URI. The assembly is imported when the stylesheet is compiled. Here is a simple representation of how the `msxsl:assembly` element is to be used.

```
<msxsl:script>
    <msxsl:assembly name="myAssembly.assemblyName" />
    <msxsl:assembly href="pathToAssembly" />

    ...

</msxsl:script>
```

The assembly name can be a full name, such as:

```
"system.Math, Version=3.1.4500.1 Culture=neutral
PublicKeyToken=a46b3f648229c514"
```

or a short name, such as `"myAssembly.Draw"`.

Namespaces

Namespaces can be declared with the **msxsl:using** element. This enables assembly classes to be written in the script without their namespaces, thus saving you some tedious typing. Here is how the `msxsl:using` element is used so as to declare namespaces.

```
<msxsl:script>
    <msxsl:using namespace="myAssemblyNS.NamespaceName" />

    ...

</msxsl:script>
```

The value of the `namespace` attribute is the name of the namespace.

3.6.4 Altova Extension Functions

Altova extension functions are in the namespace **http://www.altova.com/xslt-extensions** and are indicated in this section with the prefix **altova:** , which is assumed to be bound to the namespace given above.

The following extension functions are supported in the current version of your Altova product in the manner described below. However, note that in future versions of your product, support for one or more of these functions might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

- **altova:evaluate()**
- **altova:distinct-nodes()**
- **altova:encode-for-rtf()**
- **altova:xbrl-labels()**
- **altova:xbrl-footnotes()**

altova:evaluate()

The `altova:evaluate()` function takes an XPath expression, passed as a string, as its argument and returns the output of the evaluated expression. The XPath expression can contain variables, the values of which are passed as the subsequent arguments of the function.

```
altova:evaluate(XPathExp as xs:string [, $p1 as item()*... $pN=as item()*]) as item()*
```

The `altova:evaluate()` extension function is useful in situations where an XPath expression in the XSLT stylesheet contains one or more parts that must be evaluated dynamically. For example, consider a situation in which a user enters his request for the sorting criterion and this criterion is stored in the attribute `UserReq/@sortkey`. In the stylesheet, you could then have the expression :

```
<xsl:sort select="altova:evaluate(../UserReq/@sortkey)" order="ascending"/>
```

The `altova:evaluate()` function reads the `sortkey` attribute of the `UserReq` child element of the parent of the context node. Say the value of the `sortkey` attribute is `Price`, then `Price` is returned by the `altova:evaluate()` function and becomes the value of the `select` attribute:

```
<xsl:sort select="Price" order="ascending"/>
```

If this `sort` instruction occurs within the context of an element called `Order`, then the `Order` elements will be sorted according to the values of their `Price` children. Alternatively, if the value of `@sortkey` were, say, `Date`, then the `Order` elements would be sorted according to the values of their `Date` children. So the sort criterion for `Order` is selected from the `sortkey` attribute at runtime. This could not have been achieved with an expression like:

```
<xsl:sort select="../UserReq/@sortkey" order="ascending"/>
```

In the case shown above, the sort criterion would be the `sortkey` attribute itself, not `Price` or `Date` (or any other current content of `sortkey`).

Variables can be used in the `altova:evaluate()` extension function as shown in the examples below:

- Static variables: `<xsl:value-of select="$i3, $i2, $i1" />`
  Outputs the values of three variables.

- Dynamic XPath expression with dynamic variables:
```
<xsl:variable name="xpath" select="'$p3, $p2, $p1'" />
<xsl:value-of select="altova:evaluate( $xpath, 10, 20, 30 )" />
```
Outputs "`30 20 10`*"*

- Dynamic XPath expression with no dynamic variable:
```
<xsl:variable name="xpath" select="'$p3, $p2, $p1'" />
<xsl:value-of select="altova:evaluate( $xpath )" />
```
Outputs "`$p3, $p2, $p1`*"*

Note: The static context includes namespaces, types, and functions—but not variables—from the calling environment. The base URI and default namespace are inherited.

altova: distinct-nodes()

The `altova:distinct-modes()` function takes a set of one or more nodes as its input and returns the same set minus nodes with duplicate values. The comparison is done using the XPath/XQuery function `fn:deep-equal`.

```
altova:distinct-nodes( $arg as node()* )  as node()*
```

altova: encode-for-rtf()

The `altova:encode-for-rtf()` function converts the input string into code for RTF.

```
altova:encode-for-rtf( $inputstr as xs:string?,
$preserveallwhitespace as xs:boolean,
$preservenewlines as as xs:boolean) as xs:string
```

Whitespace and new lines will be preserved according to the boolean value specified for their respective parameters.

altova: xbrl-labels()

The `altova:xbrl-labels()` function takes two input arguments: a node name and the taxonomy file location containing the node. The function returns the XBRL labels associated with the input node.

```
altova:xbrl-labels( $name as xs:QName, $file as xs:string )  as node()*
```

altova: xbrl-footnotes()

The `altova:footnotes()` function takes a node as its input argument and returns the set of XBRL footnote nodes referenced by the input node.

```
altova:footnotes( $arg as node() )  as node()*
```

Chapter 4

License Agreement

4 License Agreement

THIS IS A LEGAL DOCUMENT -- RETAIN FOR YOUR RECORDS
ALTOVA DEVELOPER LICENSE AGREEMENT
FOR ALTOVAXML SOFTWARE

Licensor:

Altova GmbH
Rudolfsplatz 13a/9
A-1010 Wien
Austria

Important – Read Carefully. Notice to User

This Altova Developer License Agreement ("DLA") governs your right to use, bundle, integrate and distribute AltovaXML software (the "Software"). Additional information about the Software can be found on the Altova Web Site. This DLA is a legal document between you and Altova GmbH ("Altova"). It is important that you read this document before using the Altova-provided software and any accompanying documentation, including, without limitation, printed materials, 'online' files, or electronic documentation ("Documentation"). By installing the Software, or including the Software in your application, or distributing the Software, or otherwise using the Software, you agree to be bound by the terms of this DLA as well as the Altova Privacy Policy ("Privacy Policy") including, without limitation, the warranty disclaimers, limitation of liability, data use and termination provisions below. You agree that this agreement is enforceable like any written agreement negotiated and signed by you. If you do not agree, you are not licensed to use or distribute the Software, and you must destroy any downloaded copies of the Software in your possession or control. Please go to our Web site at http://www.altova.com/ALTOVAXMLdla to download and print a copy of this DLA for your files and http://www.altova.com/privacy to review the privacy policy.

1. SOFTWARE LICENSE

(a) License Grant. Upon your acceptance of this DLA, Altova grants you a non-exclusive, non-transferable limited worldwide license to: (i) develop software applications that include the Software and/or Documentation, (ii) reproduce the Software and/or Documentation, and (iii) distribute the Software in executable form and Documentation in the manner hereinafter provided to end users for the purpose of being used in conjunction with a software application developed by you.

(b) Internal Use. You may install the Software on a server within your network for the purpose of downloading and installing the Software (to an unlimited number of client computers on your internal network).

(c) External Use. You may distribute the Software and/or Documentation to any third party electronically or via download from the website or on physical media such as CD-ROMS or diskettes as part of or in conjunction with products that you have developed.

(d) Distribution Restrictions. In addition to the restrictions and obligations provided in other sections of this DLA, your license to distribute the Software and/or Documentation is further subject to all of the following restrictions: (i) the Software and/or Documentation shall only be licensed and not sold; (ii) you may not make the Software and/or Documentation available as a stand alone product and if distributed as part of a product bundle you may charge for the product bundle provided that you license such product bundle at the same or lower fee at which you license any reasonably equivalent product bundle which does not include the Software; (iii) you must use the Software and/or Documentation provided by Altova AS IS and may not impair, alter or remove Altova's copyright or license statements or any other files; and (iv) other Altova products cannot be distributed or used under this DLA.

(e) Title. This DLA gives you a limited license to reproduce and distribute the Software and/or

Documentation. Altova and its suppliers retain all right, title and interest, including all copyright and intellectual property rights, in and to, the Software and/or Documentation and all copies thereof. All rights not specifically granted in this DLA are reserved by Altova.

(f) Reverse Engineering. You may not reverse engineer, decompile, disassemble or otherwise attempt to discover the source code, underlying ideas, underlying user interface techniques or algorithms of the Software by any means whatsoever, directly or indirectly, or disclose any of the foregoing, except to the extent you may be expressly permitted to decompile under applicable law if , it is essential to do so in order to achieve operability of the Software with another software program, and you have first requested Altova to provide the information necessary to achieve such operability and Altova has not made such information available. Altova has the right to impose reasonable conditions and to request a reasonable fee before providing such information. Any information supplied by Altova or obtained by you, as permitted hereunder, may only be used by you for the purpose described herein and may not be disclosed to any third party or used to create any software which is substantially similar to the expression of the Software.

(g) Additional Restrictions. You may not loan, rent, lease, sublicense, distribute or otherwise transfer all or any portion of the Software and/or Documentation to third parties except to the limited extent expressly provided herein. You may not copy, distribute or make derivative works of the Software and/or Documentation except as expressly set forth above, and any copies that you are permitted to make pursuant to this DLA must contain the same copyright, patent and other intellectual property markings that appear on or in the Software and/or Documentation. You may not alter, modify, adapt or translate the Software and/or Documentation or any part thereof. You may not, directly or indirectly, encumber or suffer to exist any lien or security interest on the Software; knowingly take any action that would cause the Software and/or Documentation to be placed in the public domain; or use the Software and/or Documentation in any computer environment not specified in this DLA. You will comply with applicable law and Altova's instructions regarding the use of the Software and/or Documentation. You agree to notify your employees and agents who may have access to the Software and/or Documentation of the restrictions contained in this DLA and to ensure their compliance with these restrictions. You agree to indemnify, hold harmless, and defend Altova from and against any claims or lawsuits, including attorney's fees that arise or result from your use or distribution of the Software and/or Documentation.

2. INTELLECTUAL PROPERTY RIGHTS

Acknowledgement of Altova's Rights. You acknowledge that the Software and/or Documentation and any copies that you are authorized by Altova to make are the intellectual property of and are owned by Altova and its suppliers. The structure, organization and code of the Software and/or Documentation are the valuable trade secrets and confidential information of Altova and its suppliers. The Software and/or Documentation is protected by copyright, including without limitation by United States Copyright Law, international treaty provisions and applicable laws in the country in which it is being used. You acknowledge that Altova retains the ownership of all patents, copyrights, trade secrets, trademarks and other intellectual property rights pertaining to the Software and/or Documentation, and that Altova's ownership rights extend to any images, photographs, animations, videos, audio, music, text and "applets" incorporated into the Software and/or Documentation and all accompanying printed materials. You will take no actions which adversely affect Altova's intellectual property rights in the Software and/or Documentation. Trademarks shall be used in accordance with accepted trademark practice, including identification of trademark owners' names. Trademarks may only be used to identify printed output produced by the Software, and such use of any trademark does not give you any right of ownership in that trademark., XMLSPY, AUTHENTIC, STYLEVISION, MAPFORCE, SCHEMAAGENT, DIFFDOG, UMODEL MARKUP YOUR MIND, AXAD, NANONULL, and ALTOVA are trademarks and/or registered trademark of Altova GmbH. Unicode and the Unicode Logo are trademarks of Unicode, Inc. Windows, Windows 95, Windows 98, Windows NT, Windows 2000 and Windows XP are trademarks of Microsoft. W3C, CSS, DOM, MathML, RDF, XHTML, XML and XSL are trademarks (registered in numerous countries) of the World Wide Web Consortium (W3C); marks of the

W3C are registered and held by its host institutions, MIT, INRIA and Keio. Except as expressly stated above, this DLA does not grant you any intellectual property rights in the Software and/or Documentation. Notifications of claimed copyright infringement should be sent to Altova's copyright agent as further provided on the Altova Web site.

3. WARRANTY DISCLAIMER AND LIMITATION OF LIABILITY

(a) THE SOFTWARE AND/OR DOCUMENTATION ARE PROVIDED TO YOU FREE OF CHARGE, AND ON AN "AS-IS" BASIS. ALTOVA PROVIDES NO TECHNICAL SUPPORT OR WARRANTIES FOR THE SOFTWARE AND/OR DOCUMENTATION. TO THE MAXIMUM EXTENT PERMITTED BY LAW, ALTOVA AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES AND REPRESENTATIONS, WHETHER EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE; MERCHANTABILITY; SATISFACTORY QUALITY, INFORMATIONAL CONTENT, OR ACCURACY, QUIET ENJOYMENT, TITLE, AND NON- INFRINGEMENT. ALTOVA DOES NOT WARRANT THAT THE SOFTWARE IS ERROR-FREE OR WILL OPERATE WITHOUT INTERRUPTION. IF APPLICABLE LAW REQUIRES ANY WARRANTIES WITH RESPECT TO THE SOFTWARE, ALL SUCH WARRANTIES ARE LIMITED IN DURATION TO 30 DAYS FROM THE DATE OF INSTALLATION OR USE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER LEGAL RIGHTS THAT VARY FROM STATE TO STATE. YOU AGREE THAT YOU ARE SOLELY RESPONSIBLE FOR THE ACCURACY AND ADEQUACY OF THE SOFTWARE FOR YOUR INTENDED USE AND YOU WILL INDEMNIFY AND HOLD HARMLESS ALTOVA FROM ANY 3RD PARTY SUIT TO THE EXTENT BASED UPON THE ACCURACY AND ADEQUACY OF THE SOFTWARE AND/OR DOCUMENTATION IN YOUR USE. WITHOUT LIMITATION, THE SOFTWARE IS NOT INTENDED FOR USE IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE CONTROLS INCLUDING WITHOUT LIMITATION THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION, COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, LIFE SUPPORT, OR WEAPONS SYSTEMS, WHERE THE FAILURE OF THE SOFTWARE COULD LEAD TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE.

(b) TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL ALTOVA OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE AND/OR DOCUMENTATION, OR ANY PROVISION OF THIS DLA, EVEN IF ALTOVA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. WHERE LEGALLY, LIABILITY CANNOT BE EXCLUDED, BUT MAY BE LIMITED, ALTOVA'S LIABILITY AND THAT OF ITS SUPPLIERS SHALL BE LIMITED TO THE SUM OF FIVE DOLLARS (USD. $5.00) IN TOTAL. BECAUSE SOME STATES AND JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU. IN SUCH STATES AND JURISDICTIONS, ALTOVA'S LIABILITY AND THAT OF ITS SUPPLIERS SHALL BE LIMITED TO THE GREATEST EXTENT PERMITTED BY LAW. THE FOREGOING LIMITATIONS ON LIABILITY ARE INTENDED TO APPLY TO THE WARRANTIES AND DISCLAIMERS ABOVE AND ALL OTHER ASPECTS OF THIS DLA.

4. DATA USE

The terms and conditions of the Privacy Policy are set out in full at http://www.altova.com/privacy and are incorporated by reference into this DLA. By your acceptance of the terms of this DLA or use of the Software, you authorize the collection, use and

disclosure of information collected by Altova for the purposes provided for in this -DLA and/or the Privacy Policy as revised from time to time. European users understand and consent to the processing of personal information in the United States for the purposes described herein. Altova has the right in its sole discretion to amend this provision of the DLA and/or Privacy Policy at any time. You are encouraged to review the terms of the Privacy Policy as posted on the Altova Web site from time to time.

5. EXPORT RULES AND GOVERNMENT RESTRICTED RIGHTS

The Software was developed entirely at private expense and is commercial computer software provided with **RESTRICTED RIGHTS**. Use, duplication or disclosure by the U.S. Government or a U.S. Government contractor or subcontractor is subject to the restrictions set forth in this Agreement and as provided in FAR 12.211 and 12.212 (48 C.F.R. §12.211 and 12.212) or DFARS 227. 7202 (48 C.F.R. §227-7202) as applicable. Consistent with the above as applicable, Commercial Computer Software and Commercial Computer Documentation is licensed to U.S. government end users only as commercial items and only with those rights as are granted to all other end users under the terms and conditions set forth in this **DLA**. Manufacturer is Altova GmbH, Rudolfsplatz, 13a/9, A-1010 Vienna, Austria/EU. You may not use or otherwise export or re-export the Software or Documentation except as authorized by United States law and the laws of the jurisdiction in which the Software and/or Documentation was obtained. In particular, but without limitation, the Software and/or Documentation may not be exported or re-exported (i) into (or to a national or resident of) any U.S. embargoed country or (ii) to anyone on the U.S. Treasury Department's list of Specially Designated Nationals or the U.S. Department of Commerce's Table of Denial Orders. By using the Software and/or Documentation, you represent and warrant that you are not located in, under control of, or a national or resident of any such country or on any such list.

6. TERM AND TERMINATION

Without prejudice to any other rights or remedies of Altova, this DLA may be terminated (a) by you giving Altova written notice of termination; or (b) by Altova, for any or no reason, giving you written notice of termination or (c) Altova giving you written notice of termination if you fail to comply with the terms and conditions of the DLA. Upon any termination of this DLA, you must cease all use of the Software and/or Documentation, licensed hereunder, destroy all copies then in your possession or control and take such other actions as Altova may reasonably request to ensure that no copies of the Software and/or Documentation remain in your possession or control. The terms and conditions set forth in Sections 1 (e), (f), (g), 2,3, 5, 6 , and 7 survive termination of this agreement as applicable.

7. GENERAL PROVISIONS

If you are located in the European Union and are using the Software and/or Documentation in the European Union and not in the United States, then this DLA will be governed by and construed in accordance with the laws of the Republic of Austria (excluding its conflict of laws principles and the U.N. Convention on Contracts for the International Sale of Goods) and you expressly agree that exclusive jurisdiction for any claim or dispute with Altova or relating in any way to your use of the Software and/or Documentation resides in the Handelsgericht Wien (Commercial Court, Vienna) and you further agree and expressly consent to the exercise of personal jurisdiction in the Handelsgericht Wien (Commercial Court, Vienna) in connection with any such dispute or claim.

If you are located in the United States or are using the Software and/or Documentation in the United States then this DLA will be governed by and construed in accordance with the law of the Commonwealth of Massachusetts, USA (excluding its conflict of laws principles and the U.N. Convention on Contracts for the International Sale of Goods) and you expressly agree that exclusive jurisdiction for any claim or dispute with Altova or relating in any way to your use of the Software and/or Documentation resides in the federal or state courts of Massachusetts and you further agree and expressly consent to the exercise of personal jurisdiction in the federal or state courts of Massachusetts in connection with any such dispute or claim.

If you are located outside of the European Union or the United States and are not using the

Software and/or Documentation in the United States, then this DLA will be governed by and construed in accordance with the laws of the Republic of Austria (excluding its conflict of laws principles and the U.N. Convention on Contracts for the International Sale of Goods) and you expressly agree that exclusive jurisdiction for any claim or dispute with Altova or relating in any way to your use of the Software and/or Documentation resides in the Handelsgericht Wien (Commercial Court, Vienna) and you further agree and expressly consent to the exercise of personal jurisdiction in the Handelsgericht Wien (Commercial Court, Vienna) in connection with any such dispute or claim.
This DLA will not be governed by the conflict of law rules of any jurisdiction or the United Nations Convention on Contracts for the International Sale of Goods, the application of which is expressly excluded. This DLA contains the entire agreement and understanding of the parties with respect to the subject matter hereof, and supersedes all prior written and oral understandings of the parties with respect to the subject matter hereof. Any notice or other communication given under this DLA shall be in writing and shall have been properly given by either of us to the other if sent by certified or registered mail, return receipt requested, or by overnight courier to the address shown on Altova's Web site for Altova and the address shown in Altova's records for you, or such other address as the parties may designate by notice given in the manner set forth above. This DLA will bind and inure to the benefit of the parties and our respective heirs, personal and legal representatives, affiliates, successors and permitted assigns. The failure of either of us at any time to require performance of any provision hereof shall in no manner affect such party's right at a later time to enforce the same or any other term of this DLA. This DLA may be amended only by a document in writing signed by both of us. In the event of a breach or threatened breach of this DLA by either party, the other shall have all applicable equitable as well as legal remedies. Each party is duly authorized and empowered to enter into and perform this DLA. If, for any reason, any provision of this DLA is held invalid or otherwise unenforceable, such invalidity or unenforceability shall not affect the remainder of this DLA, and this DLA shall continue in full force and effect to the fullest extent allowed by law. The parties knowingly and expressly consent to the foregoing terms and conditions.

Last updated: 2005-06-28

Index

D

E

F

H

I

J

L

M

N

O

P

Q

R

S

U

V

W

X

Z