## Contents

**Preface** .......................................................................................................................................................... xxxvii
  - Documentation Accessibility .................................................................................................................... xxxvii
  - Related Documents .............................................................................................................................. xxxvii
  - Conventions ........................................................................................................................................ xxxviii

**What's New in the XDK?** ............................................................................................................................ xli
  - Features Introduced in Oracle XML Developer's Kit 11g Release 1 (11.1) ........................................ xli

1 **Introduction to Oracle XML Developer's Kit**

  - Overview of Oracle XML Developer's Kit (XDK) .............................................................................. 1-1
  - XDK Components ............................................................................................................................... 1-3
    - XML Parsers .................................................................................................................................. 1-4
    - XSLT Processors ............................................................................................................................. 1-5
    - XML Schema Processors ............................................................................................................... 1-5
    - XML Class Generators ................................................................................................................... 1-6
    - XML Pipeline Processor ................................................................................................................ 1-6
    - XDK JavaBeans ............................................................................................................................... 1-7
    - Oracle XML SQL Utility (XSU) .................................................................................................... 1-7
      - Handling or Representing an XML Document ............................................................................. 1-8
      - Using XSU with an XML Class Generator ................................................................................ 1-8
    - TransX Utility ............................................................................................................................... 1-8
    - XSQL Pages Publishing Framework .......................................................................................... 1-9
    - Soap Services ............................................................................................................................... 1-9
    - XSLT Virtual Machine (XVM) ....................................................................................................... 1-9
  - XML Document Generation with the XDK Components .................................................................... 1-10
    - XML Document Generation with Java .......................................................................................... 1-10
    - XML Document Generation with C .............................................................................................. 1-12
    - XML Document Generation with C++ ........................................................................................... 1-12
  - Development Tools and Frameworks for the XDK ............................................................................. 1-13
    - Oracle JDeveloper ........................................................................................................................ 1-14
    - User Interface XML (UIX) ............................................................................................................. 1-16
    - Oracle Reports .............................................................................................................................. 1-16
    - Oracle XML Gateway .................................................................................................................... 1-16
    - Oracle Data Provider for .NET ..................................................................................................... 1-16
  - Installing the XDK ............................................................................................................................... 1-17
Part I  XDK for Java

2  Unified Java API for XML

Overview of Unified Java API ........................................................................................................ 2-1
Component Unification .................................................................................................................. 2-1
Moving to the Unified Java API Model ....................................................................................... 2-2
    Java DOM APIs for XMLType Classes .................................................................................. 2-2
    Extension APIs ...................................................................................................................... 2-3
    Document Creation Java APIs ............................................................................................. 2-3

3  Getting Started with Java XDK Components

Installing Java XDK Components ............................................................................................... 3-1
Java XDK Component Dependencies ......................................................................................... 3-2
Setting Up the Java XDK Environment ....................................................................................... 3-5
    Setting Java XDK Environment Variables for UNIX .......................................................... 3-5
    Testing the Java XDK Environment on UNIX .................................................................. 3-5
    Setting Java XDK Environment Variables for Windows .................................................... 3-6
    Testing the Java XDK Environment on Windows ............................................................... 3-7
Verifying the Java XDK Components Version ........................................................................... 3-8

4  XML Parsing for Java

Introduction to the XML Parsing for Java .................................................................................. 4-1
    Prerequisites ......................................................................................................................... 4-1
    Standards and Specifications ............................................................................................... 4-2
    Large Node Handling .......................................................................................................... 4-2
    XML Parsing in Java ........................................................................................................... 4-3
    DOM in XML Parsing .......................................................................................................... 4-4
        DOM Creation .................................................................................................................. 4-4
    Scalable DOM ...................................................................................................................... 4-4
        Pluggable DOM Support ................................................................................................. 4-5
        Lazy Materialization ....................................................................................................... 4-5
        Configurable DOM Settings ......................................................................................... 4-5
    SAX in the XML Parser ........................................................................................................ 4-5
    JAXP in the XML Parser ....................................................................................................... 4-5
    Namespace Support in the XML Parser ............................................................................... 4-6
    Validation in the XML Parser .............................................................................................. 4-8
    Compression in the XML Parser .......................................................................................... 4-9
Using XML Parsing for Java: Overview .................................................................................... 4-9
    Using the XML Parser for Java: Basic Process ................................................................ 4-10
    Running the XML Parser Demo Programs ....................................................................... 4-10
    Using the XML Parser Command-Line Utility ................................................................ 4-13
Parsing XML with DOM .............................................................................................................. 4-13
    Using the DOM API ............................................................................................................. 4-14
    DOM Parser Architecture .................................................................................................. 4-14
        Performing Basic DOM Parsing .................................................................................... 4-15
    Useful Methods and Interfaces .......................................................................................... 4-17
5 Using Binary XML for Java

Introduction to Binary XML for Java................................................................. 5-1
Binary XML Storage Format............................................................................. 5-1
Binary XML Processors .................................................................................... 5-2
Using the JAXB Class Generator

Introduction to the JAXB Class Generator ................................................................. 8-1
  Prerequisites ............................................................................................................. 8-1
  Standards and Specifications .................................................................................. 8-1
  JAXB Class Generator Features .............................................................................. 8-2
  Marshalling and Unmarshalling with JAXB .............................................................. 8-2
  Validation with JAXB .............................................................................................. 8-3
  JAXB Customization .............................................................................................. 8-3

Using the JAXB Class Generator: Overview .......................................................... 8-4
  Using the JAXB Processor: Basic Process ............................................................... 8-4
  Running the XML Schema Processor Demo Programs ............................................. 8-7
  Using the JAXB Class Generator Command-Line Utility ..................................... 8-8
    Using oraxml to Validate Against a Schema ....................................................... 8-9
    Using oraxml to Validate Against a DTD ............................................................. 8-9
  Using the JAXB Processor: Basic Process ............................................................... 8-9
    JAXB Features Not Supported in the XDK ......................................................... 8-9

Processing XML with the JAXB Class Generator ................................................... 8-9
  Binding Complex Types ......................................................................................... 8-10

Using the Schema Processor for Java

Introduction to XML Validation ............................................................................... 7-1
  Prerequisites ............................................................................................................. 7-1
  Standards and Specifications .................................................................................. 7-1
  XML Validation with DTDs ..................................................................................... 7-2
    DTD Samples in the XDK .................................................................................... 7-2
  XML Validation with XML Schemas ..................................................................... 7-3
    XML Schema Samples in the XDK ...................................................................... 7-4
  Differences Between XML Schemas and DTDs .................................................... 7-6

Using the XML Schema Processor: Overview ....................................................... 7-7
  Using the XML Schema Processor: Basic Process ............................................... 7-7
  Running the XML Schema Processor Demo Programs .......................................... 7-9
  Using the XML Schema Processor Command-Line Utility ................................... 7-11
    Using oraxml to Validate Against a Schema ....................................................... 7-11
    Using oraxml to Validate Against a DTD ............................................................. 7-12

Validating XML with XML Schemas ..................................................................... 7-12
  Validating Against Internally Referenced XML Schemas ..................................... 7-12
  Validating AgainstExternally Referenced XML Schemas ..................................... 7-13
  Validating a Subsection of an XML Document ...................................................... 7-15
  Validating XML from a SAX Stream ..................................................................... 7-15
  Validating XML from a DOM ............................................................................... 7-17
  Validating XML from Designed Types and Elements .......................................... 7-18
  Validating XML with the XSDValidator Bean ....................................................... 7-20

Tips and Techniques for Programming with XML Schemas ................................ 7-21
  Overriding the Schema Location with an Entity Resolver .................................... 7-21
  Converting DTDs to XML Schemas ...................................................................... 7-23

8 Using the JAXB Class Generator

Introduction to the JAXB Class Generator ................................................................. 8-1
  Prerequisites ............................................................................................................. 8-1
  Standards and Specifications .................................................................................. 8-1
  JAXB Class Generator Features .............................................................................. 8-2
  Marshalling and Unmarshalling with JAXB .............................................................. 8-2
  Validation with JAXB .............................................................................................. 8-3
  JAXB Customization .............................................................................................. 8-3

Using the JAXB Class Generator: Overview .......................................................... 8-4
  Using the JAXB Processor: Basic Process ............................................................... 8-4
  Running the XML Schema Processor Demo Programs ............................................. 8-7
  Using the JAXB Class Generator Command-Line Utility ..................................... 8-8
    Using oraxml to Validate Against a Schema ....................................................... 8-9
    Using oraxml to Validate Against a DTD ............................................................. 8-9
  Using the JAXB Class Generator Command-Line Utility: Example ......................... 8-9
  JAXB Features Not Supported in the XDK .............................................................. 8-9

Processing XML with the JAXB Class Generator ................................................... 8-9
  Binding Complex Types ......................................................................................... 8-10
11 Using the XML SQL Utility (XSU)

Introduction to the XML SQL Utility (XSU) ............................................................. 11-1
  Prerequisites ............................................................................................................. 11-1
  XSU Features ............................................................................................................ 11-1
  XSU Restrictions ..................................................................................................... 11-2

Using the XML SQL Utility: Overview ................................................................. 11-2
  Using XSU: Basic Process ....................................................................................... 11-2
    Generating XML with the XSU Java API: Basic Process ................................ 11-3
    Performing DML with the XSU Java API: Basic Process ................................ 11-4
  Generating XML with the XSU PL/SQL API: Basic Process ................................ 11-6
  Performing DML with the PL/SQL API: Basic Process ........................................ 11-7

Installing XSU ........................................................................................................ 11-8
  Installing XSU in the Database ............................................................................ 11-9
  Installing XSU in an Application Server ............................................................ 11-9
  Installing XSU in a Web Server .......................................................................... 11-10

Running the XSU Demo Programs ........................................................................ 11-11
  Using the XSU Command-Line Utility ............................................................... 11-14
    Generating XML with the XSU Command-Line Utility .................................. 11-16
    Generating XMLType Data with the XSU Command-Line Utility ................ 11-16
    Performing DML with the XSU Command-Line Utility .................................. 11-16

Programming with the XSU Java API ................................................................ 11-17
  Generating a String with OracleXMLQuery ...................................................... 11-17
    Running the testXMLSQL Program ................................................................. 11-18
  Generating a DOM Tree with OracleXMLQuery .............................................. 11-18
  Paginating Results with OracleXMLQuery ....................................................... 11-19
    Limiting the Number of Rows in the Result Set .............................................. 11-19
    Keeping the Object Open for the Duration of the User’s Session ................. 11-20
    Paginating Results with OracleXMLQuery: Example .................................. 11-20
  Generating Scrollable Result Sets ................................................................. 11-21
  Generating XML from Cursor Objects ............................................................. 11-22
  Inserting Rows with OracleXMLSave ............................................................... 11-22
    Inserting XML into All Columns with OracleXMLSave ................................. 11-22
    Inserting XML into a Subset of Columns with OracleXMLSave .................. 11-23
  Updating Rows with OracleXMLSave .............................................................. 11-24

Processing XML with the XDK JavaBeans ......................................................... 11-18
  Processing XML Asynchronously with the DOMBuilder and XSLTransformer Beans .... 10-19
    Parsing the Input XSLT Stylesheet ................................................................ 10-20
    Processing the XML Documents Asynchronously ....................................... 10-21
  Comparing XML Documents with the XMLDiff Bean .................................... 10-23
    Comparing the XML Files and Generating a Stylesheet ............................... 10-24
Using the TransX Utility

Introduction to the TransX Utility
Prerequisites
TransX Utility Features
Simplified Multilingual Data Loading
Simplified Data Format Support and Interface
Additional TransX Utility Features

Using the TransX Utility: Overview
Using the TransX Utility: Basic Process
Running the TransX Utility Demo Programs
# Data Loading Format (DLF) Specification

## Introduction to DLF

- Elements and Attributes ................................................................. 13-1
- Values ................................................................................................. 13-2
- File Extensions .................................................................................... 13-2

## General Structure of DLF

- Tree Structure of DLF ........................................................................... 13-2

## DLF Specifications

- XML Declaration in DLF ......................................................................... 13-4
- Entity References in DLF ........................................................................ 13-5
- Elements in DLF ...................................................................................... 13-5
  - Top Level Table Element .................................................................. 13-5
  - Translation Elements .......................................................................... 13-6
  - Lookup Key Elements ......................................................................... 13-6
  - Metadata Elements ............................................................................. 13-6
  - Data Elements .................................................................................... 13-7
- Attributes in DLF ................................................................................... 13-7
  - DLF Attributes ................................................................................... 13-8
  - XML Namespace Attributes ............................................................... 13-10

## DLF Examples

- Minimal DLF Document ......................................................................... 13-11
- Typical DLF Document ........................................................................... 13-11
- Localized DLF Document ....................................................................... 13-13

## DLF References

- ................................................................................................................ 13-14

# Using the XSQL Pages Publishing Framework

## Introduction to the XSQL Pages Publishing Framework

- Prerequisites ......................................................................................... 14-1

## Using the XSQL Pages Publishing Framework: Overview

- Using the XSQL Pages Framework: Basic Process ................................ 14-2
- Setting Up the XSQL Pages Framework ................................................ 14-2
  - Creating and Testing XSQL Pages with Oracle JDeveloper ............... 14-5
  - Setting the CLASSPATH for XSQL Pages ........................................... 14-6
  - Configuring the XSQL Servlet Container ........................................... 14-6
15 Using the XSQL Pages Publishing Framework: Advanced Topics

Customizing the XSQL Configuration File Name .................................................. 15-1
Controlling How Stylesheets Are Processed ......................................................... 15-2
  Overriding Client Stylesheets ............................................................................. 15-2
  Controlling the Content Type of the Returned Document ................................. 15-3
  Assigning the Stylesheet Dynamically ................................................................. 15-3
  Processing XSLT Stylesheets in the Client .......................................................... 15-4
  Providing Multiple Stylesheets ............................................................................ 15-4
Working with Array-Valued Parameters ................................................................. 15-5
  Supplying Values for Array-Valued Parameters ................................................. 15-6
  Setting Array-Valued Page or Session Parameters from Strings ..................... 15-7
  Binding Array-Valued Parameters in SQL and PL/SQL Statements .................. 15-7
Setting Error Parameters on Built-in Actions ...................................................... 15-10
Using Conditional Logic with Error Parameters ................................................................. 15-10
Formatting XSQL Action Handler Errors ............................................................................. 15-11
Including XMLType Query Results in XSQL Pages ............................................................ 15-11
Handling Posted XML Content .......................................................................................... 15-14
Understanding XML Posting Options ................................................................................. 15-14
Producing PDF Output with the FOP Serializer ................................................................. 15-16
Performing XSQL Customizations ..................................................................................... 15-17
Writing Custom XSQL Action Handlers .............................................................................. 15-17
Implementing the XSQLActionHandler Interface ......................................................... 15-18
Using Multivalued Parameters in Custom XSQL Actions .................................................. 15-21
Implementing Custom XML Serializers ............................................................................. 15-21
Techniques for Using a Custom Serializer ......................................................................... 15-22
Assigning a Short Name to a Custom Serializer ............................................................... 15-22
Using a Custom XSQL Connection Manager for JDBC Datasources .............................. 15-24
Writing Custom XSQL Connection Managers ................................................................... 15-24
Accessing Authentication Information in a Custom Connection Manager ...................... 15-25
Implementing a Custom XSQLErrorHandler ...................................................................... 15-26
Providing a Custom XSQL Logger Implementation ............................................................ 15-26

Part II  XDK for C

16  Getting Started with C XDK Components

Installing C XDK Components .......................................................................................... 16-1
Configuring the UNIX Environment for C XDK Components ........................................ 16-2
C XDK Component Dependencies on UNIX ...................................................................... 16-2
Setting C XDK Environment Variables on UNIX ............................................................. 16-3
Testing the C XDK Runtime Environment on UNIX ....................................................... 16-3
Setting Up and Testing the C XDK Compile-Time Environment on UNIX ................. 16-4
  Testing the C XDK Compile-Time Environment on UNIX ........................................... 16-5
Verifying the C XDK Component Version on UNIX ......................................................... 16-5
Configuring the Windows Environment for C XDK Components ................................ 16-5
C XDK Component Dependencies on Windows............................................................... 16-5
Setting C XDK Environment Variables on Windows ....................................................... 16-6
Testing the C XDK Runtime Environment on Windows ............................................... 16-6
Setting Up and Testing the C XDK Compile-Time Environment on Windows .......... 16-7
  Testing the C XDK Compile-Time Environment on Windows ....................................... 16-7
Using the C XDK Components with Visual C/C++ on Windows ..................................... 16-8
  Setting a Path for a Project in Visual C/C++ on Windows .......................................... 16-8
  Setting the Library Path in Visual C/C++ on Windows ................................................. 16-9
Overview of the Unified C API .......................................................................................... 16-10
Globalization Support for the C XDK Components ........................................................... 16-11

17  Using the XSLT and XVM Processors for C

XVM Processor ..................................................................................................................... 17-1
XVM Usage Example ........................................................................................................... 17-1
Using the XVM Processor Command-Line Utility ............................................................ 17-3
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessing XVM Processor for C</td>
<td>17-3</td>
</tr>
<tr>
<td><strong>XML Processor</strong></td>
<td>17-3</td>
</tr>
<tr>
<td>XML Processor Usage Example</td>
<td>17-3</td>
</tr>
<tr>
<td>XPath Processor Usage Example</td>
<td>17-4</td>
</tr>
<tr>
<td>Using the C XSLT Processor Command-Line Utility</td>
<td>17-4</td>
</tr>
<tr>
<td>Accessing Oracle XSLT processor for C</td>
<td>17-5</td>
</tr>
<tr>
<td><strong>Using the Demo Files Included with the Software</strong></td>
<td>17-5</td>
</tr>
<tr>
<td>Building the C Demo Programs for XSLT</td>
<td>17-6</td>
</tr>
</tbody>
</table>

18 Using the XML Parser for C

Introduction to the XML Parser for C ............................................ 18-1
Prerequisites .................................................................................. 18-1
Standards and Specifications ....................................................... 18-2
Using the XML Parser for C ................................................................ 18-2
  Overview of the Parser API for C ................................................... 18-2
  XML Parser for C Datatypes ............................................................... 18-3
  XML Parser for C Defaults ................................................................. 18-4
  XML Parser for C Calling Sequence ...................................................... 18-4
Using the XML Parser for C: Basic Process ........................................ 18-6
Running the XML Parser for C Demo Programs ........................................ 18-7
Using the C XML Parser Command-Line Utility ....................................... 18-9
  Using the XML Parser Command-Line Utility: Example .......................... 18-10
Using the DOM API for C ...................................................................... 18-11
  Controlling the Data Encoding of XML Documents for the C API .............. 18-11
  Using NULL-Terminated and Length-Encoded C API Functions .................. 18-12
  Handling Errors with the C API ......................................................... 18-12
Using the SAX API for C ...................................................................... 18-12
What is the XML Pull Parser for C? ................................................... 18-12
  Using Basic XML Pull Parsing Capabilities ........................................ 18-13
    XML Event Context ........................................................................ 18-13
    About the XML Event Context ........................................................ 18-13
  Parsing Multiple XML Documents ....................................................... 18-14
  ID Callback ...................................................................................... 18-14
  Error Handling for the XML Pull Parser ............................................. 18-15
    Parser Errors ............................................................................... 18-15
    Programming Errors ...................................................................... 18-15
  Sample Pull Parser Application ......................................................... 18-15
    An XML Pull Parser Sample Application .......................................... 18-15
    Sample Document ......................................................................... 18-16
    Events Generated by XML Pull Parser Sample Application ................. 18-17
Using OCI and the XDK C API .............................................................. 18-17
  Using XMLType Functions and Descriptions ....................................... 18-17
  Initializing an XML Context for XML DB ........................................... 18-18
  Creating XMLType Instances on the Client ....................................... 18-18
  Operating on XML Data in the Database Server .................................. 18-19
  Using OCI and the XDK C API: Examples ........................................ 18-19
19 Using Binary XML for C

Introduction to Binary XML for C ................................................................. 19-1
Prerequisites ............................................................................................... 19-1
Binary XML Storage Format ..................................................................... 19-1

20 Using the XML Schema Processor for C

Oracle XML Schema Processor for C .......................................................... 20-1
Oracle XML Schema for C Features ......................................................... 20-1
Standards Conformance .......................................................................... 20-2
XML Schema Processor for C: Supplied Software .................................. 20-2
XML Schema Processor for C Usage Diagram ....................................... 20-3
How to Run XML Schema Processor for C Sample Programs ................. 20-3
What is the Streaming Validator?............................................................... 20-4
Using Transparent Mode ......................................................................... 20-4
   Error Handling in Transparent Mode ................................................... 20-5
   Streaming Validator Example .............................................................. 20-5
Using Opaque Mode .................................................................................. 20-6
   Error Handling in Opaque Mode ......................................................... 20-6
   Example of Opaque Mode Application ................................................ 20-6
Enhancement of the Existing XmlSchemaLoad() Function ..................... 20-7
Validation Options .................................................................................... 20-8

21 Determining XML Differences Using C

Overview of XMLDiff in C .......................................................... 21-1
   Flow of Process for XMLDiff ................................................................. 21-1
Using XMLDiff .......................................................................................... 21-2
   User Options for Optimization .......................................................... 21-2
   User Option for Hashing ....................................................................... 21-2
   How XmlDiff Looks at Input Documents ........................................... 21-2
   Using the XMLDiff Command Line Utility ....................................... 21-2
Sample Input Document .......................................................................... 21-3
Sample Xdiff Instance Document ........................................................... 21-4
Output Model and XML Processing Instructions ...................................... 21-5
Xdiff Operations ....................................................................................... 21-5
   Format of Xdiff Instance Document ................................................... 21-6
   Xdiff Schema ....................................................................................... 21-6
   Using XMLDiff in an Application ....................................................... 21-8
   Customized Output ............................................................................ 21-10
Using XmlPatch ....................................................................................... 21-11
   Using the XmlPatch Command Line Utility ....................................... 21-11
   Using XmlPatch in an Application ..................................................... 21-11
Using XmlHash ....................................................................................... 21-12
   Calling XmlDiff and XmlPatch .......................................................... 21-13
22 Using SOAP with the C XDK

Introduction to SOAP for C ................................................................. 22-1
SOAP Messaging Overview ................................................................. 22-2
SOAP Message Format ........................................................................ 22-2
Using SOAP Clients ............................................................................ 22-4
Using SOAP Servers ........................................................................... 22-4
SOAP C Functions................................................................................. 22-5
SOAP Example 1: Sending an XML Document ....................................... 22-6
SOAP Example 2: A Response Asking for Clarification ......................... 22-12
SOAP Example 3: Using POST ............................................................. 22-14

Part III Oracle XDK for C++

23 Getting Started with C++ XDK Components

Installing the C++ XDK Components .................................................. 23-1
Configuring the UNIX Environment for C++ XDK Components ............ 23-1
C++ XDK Component Dependencies on UNIX ..................................... 23-1
Setting C++ XDK Environment Variables on UNIX .............................. 23-2
Testing the C++ XDK Runtime Environment on UNIX ......................... 23-2
Setting Up and Testing the C++ XDK Compile-Time Environment on UNIX 23-2
  Testing the C++ XDK Compile-Time Environment on UNIX .......... 23-2
Verifying the C++ XDK Component Version on UNIX ........................... 23-2
Configuring the Windows Environment for C++ XDK Components ........ 23-3
C++ XDK Component Dependencies on Windows .................................. 23-3
Setting C++ XDK Environment Variables on Windows ........................ 23-3
Testing the C++ XDK Runtime Environment on Windows .................. 23-3
Setting Up and Testing the C++ XDK Compile-Time Environment on Windows 23-3
  Testing the C++ XDK Compile-Time Environment on Windows .... 23-3
Using the C++ XDK Components with Visual C/C++ .......................... 23-4

24 Overview of the Unified C++ Interfaces

What is the Unified C++ API? ............................................................... 24-1
Accessing the C++ Interface ................................................................. 24-1
OracleXML Namespace ...................................................................... 24-2
  OracleXML Interfaces ..................................................................... 24-2
Ctx Namespace .................................................................................. 24-2
  OracleXML Datatypes ................................................................... 24-2
  Ctx Interfaces .............................................................................. 24-2
IO Namespace .................................................................................... 24-3
  IO Datatypes ............................................................................... 24-3
  IO Interfaces ............................................................................. 24-3
Tools Package .................................................................................... 24-3
  Tools Interfaces ........................................................................... 24-4
Error Message Files ............................................................................ 24-4
25 Using the XML Parser for C++

Introduction to Parser for C++ ........................................................................................................ 25-1
DOM Namespace .......................................................................................................................... 25-2
  DOM Datatypes ......................................................................................................................... 25-2
  DOM Interfaces ......................................................................................................................... 25-2
  DOM Traversal and Range Datatypes ...................................................................................... 25-3
  DOM Traversal and Range Interfaces ...................................................................................... 25-3
Parser Namespace .................................................................................................................... 25-4
  GParser Interface .................................................................................................................. 25-4
  DOMParser Interface .............................................................................................................. 25-4
  SAXParser Interface .............................................................................................................. 25-4
    SAX Event Handlers ............................................................................................................. 25-4
Thread Safety ................................................................................................................................ 25-4
XML Parser for C++ Usage ........................................................................................................ 25-4
XML Parser for C++ Default Behavior .................................................................................... 25-4
C++ Sample Files ...................................................................................................................... 25-5

26 Using the XSLT Processor for C++

Accessing XSLT for C++ ............................................................................................................ 26-1
Xsl Namespace .......................................................................................................................... 26-1
  Xsl Interfaces .......................................................................................................................... 26-1
XSLT for C++ DOM Interface Usage ........................................................................................ 26-2
Invoking XSLT for C++ ............................................................................................................ 26-2
  Command Line Usage ............................................................................................................ 26-2
    Writing C++ Code to Use Supplied APIs ........................................................................... 26-2
Using the Sample Files Included with the Software ................................................................... 26-2

27 Using the XML Schema Processor for C++

Oracle XML Schema Processor for C++ .................................................................................... 27-1
  Oracle XML Schema for C++ Features ................................................................................ 27-1
    Online Documentation ........................................................................................................ 27-1
    Standards Conformance ..................................................................................................... 27-2
XML Schema Processor API .................................................................................................... 27-2
  Invoking XML Schema Processor for C++ ......................................................................... 27-2
Running the Provided XML Schema for C++ Sample Programs ........................................... 27-2

28 Using the XPath Processor for C++

XPath Interfaces ....................................................................................................................... 28-1
Sample Programs ...................................................................................................................... 28-1

29 Using the XML Class Generator for C++

Accessing XML C++ Class Generator ..................................................................................... 29-1
Using XML C++ Class Generator ............................................................................................ 29-1
  External DTD Parsing ......................................................................................................... 29-1
  Error Message Files ............................................................................................................. 29-1
Using the XML C++ Class Generator Command-Line Utility .......................................................... 29-2
Input to the XML C++ Class Generator ......................................................................................... 29-2
Using the XML C++ Class Generator Examples ........................................................................... 29-2
XML C++ Class Generator Example 1: XML — Input File to Class Generator, CG.xml ........ 29-3
XML C++ Class Generator Example 2: DTD — Input File to Class Generator, CG.dtd ........... 29-3
XML C++ Class Generator Example 3: CG Sample Program ....................................................... 29-3

Part IV Oracle XDK Reference

30 XSQL Pages Reference
XSQL Configuration File Parameters .............................................................................................. 30-2
<xsql:action> ............................................................................................................................. 30-6
<xsql:delete-request> ............................................................................................................... 30-8
<xsql:dml> ............................................................................................................................... 30-10
<xsql:if-param> ...................................................................................................................... 30-11
<xsql:include-owa> ............................................................................................................... 30-13
<xsql:include-param> ............................................................................................................ 30-15
<xsql:include-posted-xml> .................................................................................................... 30-16
<xsql:include-request-params> .............................................................................................. 30-17
<xsql:include-xml> ................................................................................................................ 30-19
<xsql:include-xsql> .............................................................................................................. 30-20
<xsql:insert-param> ................................................................................................................ 30-22
<xsql:insert-request> ............................................................................................................. 30-24
<xsql:query> .......................................................................................................................... 30-26
<xsql:ref-cursor-function> ...................................................................................................... 30-29
<xsql:set-cookie> .................................................................................................................. 30-31
<xsql:set-page-param> ......................................................................................................... 30-33
<xsql:set-session-param> ..................................................................................................... 30-36
<xsql:set-stylesheet-param> ................................................................................................. 30-38
<xsql:update-request> ............................................................................................................ 30-40

31 XDK Standards
XML Standards Supported by the XDK ........................................................................................ 31-1
Summary of XML Standards Supported by the XDK ................................................................. 31-1
XML Standards for the XDK for Java ......................................................................................... 31-2
DOM Standard for the XDK for Java ......................................................................................... 31-2
XSLT Standard for the XDK for Java ......................................................................................... 31-3
JAXB Standard for the XDK for Java ....................................................................................... 31-5
Pipeline Definition Language Standard for the XDK for Java ............................................. 31-5
Character Sets Supported by the XDK ....................................................................................... 31-5
Character Sets Supported by the XDK for Java ................................................................. 31-5
Character Sets Supported by the XDK for C .......................................................................... 31-6

A Oracle XDK for Java Error Messages
XML Error Messages .................................................................................................................. A-1
XML Parser Error Messages ...................................................................................................... A-1
XML-20190: Whitespace required. ................................................................. A-13
XML-20191: ‘>’ required to end DTD. .......................................................... A-13
XML-20192: Unexpected text in DTD. ........................................................ A-13
XML-20193: Unexpected EOF. ................................................................. A-13
XML-20194: Unable to write to output stream ........................................ A-13
XML-20195: Encoding not supported in PrintWriter. .......................... A-13
XML-20200: Expected string instead of string. ........................................ A-13
XML-20201: Expected string instead of string. ........................................ A-13
XML-20202: Expected string to be string. ............................................... A-13
XML-20205: Expected string. ................................................................. A-13
XML-20206: Expected string or string. .................................................. A-13
XML-20210: Unexpected string. ............................................................ A-14
XML-20211: string is not allowed in string. ............................................ A-14
XML-20220: Invalid InputSource. .......................................................... A-14
XML-20221: Invalid char in text.............................................................. A-14
XML-20230: Illegal change of encoding; from string to string. .......... A-14
XML-20231: Encoding string is not currently supported. ..................... A-14
XML-20240: Unable to open InputSource. ............................................. A-14
XML-20241: Unable to open entity string. ............................................. A-14
XML-20242: Error opening external DTD string. .................................... A-14
XML-20250: Missing entity string. ........................................................ A-14
XML-20251: Cyclic Entity Reference in entity string. ............................ A-14
XML-20280: Bad character (string). ........................................................ A-14
XML-20281: NMToken must contain atleast one NMChar. ................... A-14
XML-20282: string not allowed in a PubIdLiteral. ................................. A-14
XML-20284: Illegal white space before optional character in content model. A-14
XML-20285: Illegal mixed content model. ............................................. A-14
XML-20286: Content list not allowed inside optional mixed content model. A-15
XML-20287: Content particles not allowed inside optional mixed content model A-15
XML-20288: Invalid default declaration in attribute declaration. .......... A-15
XML-20500: SAX feature string not recognized. .................................... A-15
XML-20501: SAX feature string not supported. ..................................... A-15
XML-20502: SAX property string not recognized. .................................. A-15
XML-20503: SAX property string not supported. .................................... A-15

DOM Error Messages ............................................................................ A-15
XML-21000: invalid size string specified ............................................. A-15
XML-21001: invalid index string specified; must be between 0 and string A-15
XML-21002: cannot add an ancestor as a child node. ........................... A-15
XML-21003: node of type string cannot be added to node of type string A-15
XML-21004: document node can have only one string node as child .... A-15
XML-21005: node of type string cannot be added to attribute list .......... A-16
XML-21006: cannot add a node belonging to a different document ...... A-16
XML-21007: invalid character string in name ....................................... A-16
XML-21008: cannot set value for node of type string. .......................... A-16
XML-21009: cannot modify descendants of entity or entity reference nodes A-16
XML-21010: cannot modify DTD’s content ............................................ A-16
XML-21011: cannot remove attribute; not found in the current element A-16
XML-21012: cannot remove or replace node; it is not a child of the current node

XML-21013: parameter string not recognized

XML-21014: value string of parameter string is not supported

XML-21015: cannot add attribute belonging to another element

XML-21016: invalid namespace string for prefix string

XML-21017: invalid qualified name: string

XML-21018: conflicting namespace declarations string and string for prefix string

XML-21019: string object is detached

XML-21020: bad boundary specified; cannot partially select a node of type string

XML-21021: node of type string does not support range operation string

XML-21022: invalid event type: string

XML-21023: prefix not allowed on nodes of type string

XML-21024: import not allowed on nodes of type string

XML-21025: rename not allowed on nodes of type string

XML-21026: Unrepresentable character in node: string

XML-21027: Namespace normalization error in node: string

XML-21997: function not supported on THICK DOM

XML-21998: system error occurred: string

XSL Transformation Error Messages

XML-22000: Error while parsing XSL file (string)

XML-22001: XSL Stylesheet does not belong to XSLT namespace

XML-22002: Error while processing include XSL file (string)

XML-22003: Unable to write to output stream (string)

XML-22004: Error while parsing input XML document (string)

XML-22005: Error while reading input XML stream (string)

XML-22006: Error while reading input XML URL (string)

XML-22007: Error while reading input XML reader (string)

XML-22008: NameSpace prefix string used but not declared

XML-22009: Attribute string not found in string

XML-22010: Element string not found in string

XML-22011: Cannot construct XML PI with content: string

XML-22012: Cannot construct XML comment with content: string

XML-22013: Error in expression: string

XML-22014: Expecting node-set before relative location path

XML-22015: Function string not found

XML-22016: Extension function namespace should start with string

XML-22017: Literal expected in string function. Found string

XML-22018: Parse Error in string function

XML-22019: Expected string instead of string

XML-22020: Error in extension function arguments

XML-22021: Error parsing external document: string

XML-22022: Error while testing predicates. Not a nodeset type

XML-22023: Literal Mismatch

XML-22024: Unknown multiply operator

XML-22025: Expression error: Empty string

XML-22026: Unknown expression at EOF: string

XML-22027: Closing } not found in Attribute Value template
XML-22028: Expression value type string not recognized by string ........................................ A-20
XML-22029: Cannot transform child string in string ................................................................. A-20
XML-22030: Attribute value string not expected for string ...................................................... A-20
XML-22031: Variable not defined: string ................................................................................ A-20
XML-22032: Found a single ) outside expression in Attribute value template ......................... A-20
XML-22033: Token not recognized: ! ...................................................................................... A-20
XML-22034: Namespace definition not found for prefix string ................................................ A-20
XML-22035: Axis string not found ........................................................................................... A-20
XML-22036: Cannot convert string to string ............................................................................. A-21
XML-22037: Unsupported feature: string ................................................................................ A-21
XML-22038: Expected Node-set in Path Expression ................................................................ A-21
XML-22039: Extension function error: Error invoking constructor for string .......................... A-21
XML-22040: Extension function error: Overloaded constructors for string .............................. A-21
XML-22041: Extension function error: Constructor not found for string ..................................... A-21
XML-22042: Extension function error: Overloaded method string ............................................. A-21
XML-22043: Extension function error: Method not found string ................................................ A-21
XML-22044: Extension function error: Error invoking string:string ......................................... A-21
XML-22045: Extension function error: Class not found string ................................................... A-21
XML-22046: Apply import cannot be called when current template is null .............................. A-21
XML-22047: Invalid instantiation of string in string context .................................................... A-21
XML-22048: The string element children must precede all other element children of an string element. ........................................................................................................... A-21
XML-22049: Template string invoked but not defined .............................................................. A-21
XML-22050: Duplicate variable string definition ........................................................................ A-21
XML-22051: only a literal or a reference to a variable or parameter is allowed in id() function when used as a pattern  A-22
XML-22052: no sort key named as: string was defined ............................................................ A-22
XML-22053: cannot detect encoding in unparsed-text(), please specify .................................... A-22
XML-22054: no such xsl:function with namespace: string and local name: string was defined A-22
XML-22055: range expression can only accept xs:integer data type, but not string ....... A-22
XML-22056: exactly one of four group attributes must be present in xsl:for-each-group ...... A-22
XML-22057: string can only have string as children ................................................................. A-22
XML-22059: wrong child order of xsl:function ......................................................................... A-22
XML-22060: TERMINATE PROCESSING.................................................................................. A-22
XML-22061: terminate attribute in <xsl:message> can only be yes or no ............................ A-22
XML-22062: string must have at least one string child ............................................................ A-22
XML-22063: no definition for character-map with qname string ............................................. A-22
XML-22064: cannot define character-map with the same name string and the same import precedence A-22
XML-22065: at least one string must be defined under string ................................................ A-23
XML-22066: if select attribute is present, string instructions sequence-constructor must be empty A-23
XML-22067: if use attribute is present, string instructions sequence-constructor must be empty A-23
XML-22068: only primary sort key is allowed to have the stable attribute .............................. A-23
XML-22069: only string or string is allowed................................................................. A-23
XML-22101: DOMSource node as this type not supported............................................. A-23
XML-22103: DOMResult can not be this kind of node................................................. A-23
XML-22106: Invalid StreamSource - InputStream, Reader, and SystemId are null...... A-23
XML-22107: Invalid SAXSource - InputSource is null............................................... A-23
XML-22108: Invalid Source - URL format is incorrect................................................. A-23
XML-22109: Internal error while reporting SAX events............................................... A-23
XML-22110: Invalid StreamResult set in TransformerHandler................................. A-23
XML-22111: Invalid Result set in TransformerHandler............................................... A-24
XML-22112: Namespace URI missing } ....................................................................... A-24
XML-22113: Namespace URI should start with {.......................................................... A-24
XML-22117: URL format has problems (null or bad format or missing 'href' or missing '=').
A-24
XML-22121: Could not get associated stylesheet ......................................................... A-24
XML-22122: Invalid StreamResult - OutputStream, Writer, and SystemId are null...... A-24
XML-22900: An internal error condition occurred......................................................... A-24
XPath Error Messages................................................................................................. A-24
XML-23002: internal xpath error.................................................................................. A-24
XML-23003: XPath 2.0 feature schema-element/schema-attribute not supported .......... A-24
XML-23006: value does not match required type.......................................................... A-24
XML-23007: FOAR0001: division by zero..................................................................... A-24
XML-23008: FOAR0002: numeric operation overflow/unflow...................................... A-24
XML-23009: FOCA0001: Error in casting to decimal..................................................... A-25
XML-23010: FOCA0002: invalid lexical value............................................................... A-25
XML-23011: FOCA0003: input value too large for integer............................................ A-25
XML-23012: FOCA0004: Error in casting to integer..................................................... A-25
XML-23013: FOCA0005: NaN supplied as float/double value....................................... A-25
XML-23014: FOCH0001: invalid codepoint................................................................. A-25
XML-23015: FOCH0002: unsupported collation.......................................................... A-25
XML-23016: FOCH0003: unsupported normalization form......................................... A-25
XML-23017: FOCH0004: collation does not support collation units............................ A-25
XML-23018: FODC0001: no context document............................................................ A-25
XML-23019: FODC0002: Error retrieving resource...................................................... A-25
XML-23020: FODC0003: Error parsing contents of resource........................................ A-26
XML-23021: FODC0004: invalid argument to fn:collection()...................................... A-26
XML-23022: FODT0001: overflow in date/time arithmetic............................................ A-26
XML-23023: FODT0002: overflow in duration arithmetic............................................. A-26
XML-23024: FONC0001: undefined context item......................................................... A-26
XML-23025: FONS0002: default namespace is defined............................................... A-26
XML-23026: FONS0003: no prefix defined for namespace........................................... A-26
XML-23027: FONS0004: no namespace found for prefix............................................ A-26
XML-23028: FONS0005: base URI not defined in the static context............................. A-26
XML-23029: FORG0001: invalid value for cast/constructor......................................... A-26
XML-23030: FORG0002: invalid argument to fn:resolve-uri()..................................... A-26
XML-23031: FORG0003: zero-or-one called with sequence containing more than one item .. A-27
XML-23032: FORG0004: fn:one-or-more called with sequence containing no items..... A-27
XML-23033: FORG0005: exactly-one called with sequence containing zero or more than one item  A-27  
XML-23034: FORG0006: invalid argument type .................................................................  A-27  
XML-23035: FORG0007: invalid argument to aggregate function ...................................  A-27  
XML-23036: FORG0008: both arguments to fn:dateTime have a specified timezone ...  A-27  
XML-23037: FORG0009: base uri argument to fn:resolve-uri is not an absolute URI ...  A-27  
XML-23038: FORX0001: invalid regular expression flags ..................................................  A-27  
XML-23039: FORX0002: invalid regular expression ..........................................................  A-27  
XML-23040: FORX0003: regular expression matches zero-length string .........................  A-27  
XML-23041: FORX0004: invalid replacement string .........................................................  A-28  
XML-23042: FOTY0001: type error .....................................................................................  A-28  
XML-23043: FOTY0011: context item is not a node ...........................................................  A-28  
XML-23044: FOTY0012: items not comparable ...............................................................  A-28  
XML-23045: FOTY0013: type does not have equality defined .........................................  A-28  
XML-23046: FOTY0014: type exception ..........................................................................  A-28  
XML-23047: FORT0001: invalid number of parameters ..................................................  A-28  
XML-23048: FOTY0002: type definition not found ..........................................................  A-28  
XML-23049: FOTY0021: invalid node type .....................................................................  A-28  
XML-23050: FOER0000: unidentified error .....................................................................  A-28  
XML-23051: FODC0005: invalid argument to fn:doc ......................................................  A-28  
XML-23052: FODT0003: invalid timezone value .............................................................  A-29  

XML Schema Validation Error Messages ...........................................................................  A-29  
XML-24000: internal error ..............................................................................................  A-29  
XML-24001: attribute string not expected at line string, column string ..........................  A-29  
XML-24002: can not find element declaration string ......................................................  A-29  
XML-24003: context-determined element declaration string absent ................................  A-29  
XML-24004: declaration for element string absent ........................................................  A-29  
XML-24005: element string not assessed .......................................................................  A-29  
XML-24006: element string laxly assessed .....................................................................  A-29  
XML-24007: missing attribute declaration string in element string ...............................  A-29  
XML-24008: type absent for attribute string ....................................................................  A-29  
XML-24009: invalid attribute value string .....................................................................  A-30  
XML-24010: attribute value string and fixed value string not match ..............................  A-30  
XML-24011: type of element string is abstract ...............................................................  A-30  
XML-24012: no children allowed for element string with empty content type ..............  A-30  
XML-24013: element child string not allowed for simple content ...................................  A-30  
XML-24014: characters string not allowed for element-only content ..............................  A-30  
XML-24015: multiple ID attributes in element string at line string, column string ........  A-30  
XML-24016: invalid string value string at line string, column string ..............................  A-30  
XML-24017: invalid boolean value string at line string, column string ..........................  A-30  
XML-24018: invalid decimal value string at line string, column string ..........................  A-30  
XML-24019: invalid float value string at line string, column string ...............................  A-31  
XML-24020: invalid double value string at line string, column string ............................  A-31  
XML-24021: invalid duration string at line string, column string ..................................  A-31  
XML-24022: invalid date value string at line string, column string ...............................  A-31  
XML-24023: invalid dateTime value string at line string, column string ..........................  A-31  
XML-24024: invalid time value string at line string, column string ...............................  A-31  
XML-24025: invalid gYearMonth value string at line string, column string ..................  A-31  

xxvi
XML-24073: target node set not equals to qualified node set for key string .......... A-36
XML-24074: element member string in key sequence is nillable.......................... A-36
XML-24075: missing key sequence for key reference string ................................. A-36
XML-24076: incorrect length of value string ...................................................... A-36
XML-24077: value string greater than or equal to maxExclusive .......................... A-36
XML-24078: value string greater than the maxInclusive ..................................... A-37
XML-24079: value length of string greater than maxLength .................................. A-37
XML-24080: value string smaller or equals to minExclusive ................................ A-37
XML-24081: value string smaller than minInclusive .......................................... A-37
XML-24082: value string shorter than minLength .............................................. A-37
XML-24083: wildcard particle in the content of element string not done ............... A-37
XML-24084: element particle string not done .................................................... A-37
XML-24085: model group string in the content of element string not done .......... A-37
XML-24086: invalid literal string with respect to pattern facet string .................. A-37
XML-24087: undefined type string .................................................................... A-37
XML-24088: undeclared attribute string ........................................................... A-38
XML-24089: undeclared element string .............................................................. A-38
XML-24090: undefined attribute group string .................................................. A-38
XML-24091: undefined model group string ....................................................... A-38
XML-24092: undeclared notation string ............................................................. A-38
XML-24093: too many digits in value string at line string, column string .......... A-38

Schema Representation Constraint Error Messages ............................................. A-38
XML-24100: element string must belong to XML Schema namespace ............... A-38
XML-24101: can not build schema from location string ....................................... A-38
XML-24102: can not resolve schema by target namespace string ......................... A-38
XML-24103: invalid annotation representation at line string, column string ........ A-39
XML-24104: multiple annotations at line string, column string .......................... A-39
XML-24105: annotation must be the first element at line string, column string .... A-39
XML-24106: attribute wildcard before attribute declaration at line string, column string ...... A-39
XML-24107: multiple attribute wildcard ............................................................. A-39
XML-24108: default string and fixed string both present .................................... A-39
XML-24109: default value string conflicts with attribute use string XML-24109: default value string conflicts with attribute use string A-39
XML-24110: missing name or ref attribute ......................................................... A-39
XML-24111: both name and ref presented in attribute declaration ......................... A-39
XML-24112: ref conflicts with form, type, or simpleType child ............................ A-39
XML-24113: type attribute conflicts with simpleType child ................................ A-40
XML-24114: intersecton of attribute wildcard is not expressible ........................... A-40
XML-24115: circular attribute group reference string ......................................... A-40
XML-24116: circular group reference string ....................................................... A-40
XML-24117: base type string for complexContent is not complex type ................. A-40
XML-24118: simple content required in base type string ..................................... A-40
XML-24119: properties specified with element reference string .......................... A-40
XML-24120: simpleType and complexType can not both present in element declaration string A-40
XML-24121: imported namespace string must different from namespace string .... A-40
XML-24122: target namespace string required .................................................. A-41
XML-24123: namespace string is different from expected targetNamespace string .... A-41
XML-24124: targetNamespace string not expected in schema ........................................ A-41
XML-24125: can not include schema from string .......................................................... A-41
XML-24126: included targetNamespace string must the same as string ......................... A-41
XML-24127: no-namespace schema can not include schema with target namespace string ... A-41
XML-24128: itemType attribute conflicts with simpleType child ................................. A-41
XML-24129: prefix of qname string can not be resolved .............................................. A-41
XML-24130: redefined schema has different namespace, line string column string...... A-41
XML-24131: no-namespace schema can only redefine schema without targetNamespace..... A-42
XML-24132: type derivation string must be restriction ................................................. A-42
XML-24132: type string must redefine itself at line string, column string ................... A-42
XML-24133: group string can have only one self reference in redefinition .................... A-42
XML-24134: self reference of group string must not have minOccurs or maxOccurs other than 1 in redefinition A-42
XML-24135: redefined group string is not a restriction of its orginal group ................. A-42
XML-24236: attribute group string can have only one self reference in redefinition..... A-42
XML-24136: redefined attribute group string must be a restriction of its orginal group ... A-42
XML-24137: restriction must not have both base and simpleType child ........................ A-42
XML-24138: simple type restriction must have either base attribute or simpleType child ... A-43
XML-24139: neither itemType or simpleType child present for list ................................ A-43
XML-24140: itemType and simpleType child can not both be present in list type ......... A-43
XML-24141: circular union type is disallowed .............................................................. A-43
XML-24142: facet string can not be specified more than once ..................................... A-43
XML-24143: memberTypes and simpleType child can not both be absent in union ....... A-43
XML-24144: facets can only used for restriction ............................................................. A-43

Schema Component Constraint Error Messages .......................................................... A-43

XML-24201: duplicate attribute string declaration ...................................................... A-43
XML-24202: more than one attributes with ID type not allowed .................................. A-44
XML-24203: invalid value constraint string ................................................................ A-44
XML-24204: value constraint string not allowed for ID type ...................................... A-44
XML-24205: fixed value string does not match string in attribute declaration ............. A-44
XML-24206: value constraint must be fixed to match that in attribute declaration ....... A-44
XML-24207: invalid xpath expression string ............................................................... A-44
XML-24208: invalid field xpath string ...................................................................... A-44
XML-24209: maxOccurs in element string of All group must be 0 or 1 ....................... A-44
XML-24210: All group has to form a content type ....................................................... A-44
XML-24211: All group has to form a content type ....................................................... A-44
XML-24212: type string does not allow facet string .................................................. A-45
XML-24213: wildcard intersection is not expressible ................................................... A-45
XML-24214: base type not allow string derivation ....................................................... A-45
XML-24215: complex type string is not a derivation of type string ............................. A-45
XML-24216: must specify a particle in extended content type ..................................... A-45
XML-24217: content type string conflicts with base type's content type string .......... A-45
XML-24218: inconsistent local element declarations string ........................................ A-45
XML-24219: element *string* is not valid substitutable for element *string* ........................................ A-45
XML-24220: itemType *string* can not be list .......................................................................................... A-45
XML-24221: circular union *string* not allowed ...................................................................................... A-46
XML-24222: ambiguous particles *string* ............................................................................................... A-46
XML-24223: invalid particle extension .................................................................................................... A-46
XML-24224: invalid particle restriction ...................................................................................................... A-46
XML-24225: simple type *string* does not allowed restriction ................................................................ A-46
XML-24226: invalid derivation from base type *string* .......................................................................... A-46
XML-24227: atomic type can not restrict list *string* ................................................................................. A-46
XML-24228: base type can not be ur-type in restriction .......................................................................... A-46
XML-24229: base type of list must be list or ur-type .................................................................................. A-46
XML-24230: base type of union must be union or ur-type ......................................................................... A-46
XML-24231: element default *string* requires mixed content to be emptiable ................................... A-46
XML-24232: element default *string* requires mixed content or simple content .................................. A-46
XML-24233: element default *string* must be valid to its content type .................................................. A-47
XML-24234: wrong field cardinality for keyref *string* .......................................................................... A-47
XML-24235: complex type can only extend simple type *string* .............................................................. A-47
XML-24236: circular type definition *string* ......................................................................................... A-47
XML-24237: base type *string* must be complex type ............................................................................. A-47
XML-24238: attribute *string* not allowed in base type ........................................................................ A-47
XML-24239: required attribute *string* not in restriction ...................................................................... A-47
XML-24240: no correspoonding attribute wildcard in bas type *string* ................................................ A-47
XML-24241: base type *string* must have simple content or emptiable ............................................... A-47
XML-24242: base type *string* must have empty content or emptiable ............................................... A-48
XML-24243: enumeration facet required for NOTATION ......................................................................... A-48
XML-24244: invalid value *string* in enumeration .................................................................................. A-48
XML-24245: default value *string* is element type invalid ................................................................. A-48
XML-24246: invalid substitutionGroup *string*, type invalid ............................................................... A-48
XML-24247: ID type does not allow value constraint *string* ............................................................... A-48
XML-24248: fractionDigits *string* greater than totalDigits *string* ...................................................... A-48
XML-24249: length facet can not be specified with minLength or maxLength ........................................ A-48
XML-24250: length *string* not the same as length in base type’s ............................................................ A-48
XML-24251: maxExclusive greater than its original ................................................................................. A-49
XML-24252: minInclusive greater than or equal to maxExclusive .......................................................... A-49
XML-24253: maxLength is greater than that in base type .......................................................................... A-49
XML-24254: circular group *string* disallowed ...................................................................................... A-49
XML-24256: minExclusive must be less than or equal to maxExclusive ............................................... A-49
XML-24257: minExclusive *string* must be less than maxInclusive ....................................................... A-49
XML-24258: invalid minExclusive *string* ............................................................................................. A-49
XML-24259: invalid minExclusive *string* ............................................................................................. A-49
XML-24260: invalid minExclusive *string* ............................................................................................. A-49
XML-24261: invalid minExclusive *string* ............................................................................................. A-49
XML-24262: minInclusive *string* must not be greater than maxInclusive ............................................. A-50
XML-24263: Can not specify both minInclusive and minExclusive ......................................................... A-50
XML-24264: invalid minInclusive *string* ............................................................................................. A-50
XML-24265: invalid minInclusive *string* ............................................................................................. A-50
XML-24266: invalid minInclusive *string* ............................................................................................. A-50

XXX
XML-24268: invalid minInclusive string ................................................................. A-50
XML-24269: invalid minLength string ................................................................. A-50
XML-24270: invalid minLength string ................................................................. A-50
XML-24271: can not declare xmlns attribute........................................................ A-50
XML-24272: no xsi for targetNamespace................................................................. A-51
XML-24272: minOccurs is greater than max_occurs........................................... A-51
XML-24281: maxOccurs must greater than or equal to 1...................................... A-51
XML-24282: incorrect Notation properties........................................................... A-51
XML-24283: particle's range is not valid restriction.............................................. A-51
XML-24284: sequence group is not valid derivation of choice group.................. A-51
XML-24285: element string is not valid restriction of element string ................. A-51
XML-24286: element string is not valid restriction of wildcard .......................... A-51
XML-24287: group is not valid restriction of wildcard ........................................ A-51
XML-24288: group any is not valid restriction..................................................... A-51
XML-24289: invalid restriction of all or sequence group.................................... A-51
XML-24290: wildcard is not valid restriction....................................................... A-52
XML-24291: sequence is not a valid restriction of all........................................ A-52
XML-24292: duplicate component definitions string ........................................ A-52
XML-24293: Incorrect simple type definition properties..................................... A-52
XML-24294: wildcard is not a subset of its super............................................. A-52
XML-24295: totalDigits string is greater than string in base type...................... A-52
XML-24296: whiteSpace string can not restrict base type's string ...................... A-52
XML-24297: circular substitution group string ................................................ A-52

XSQL Server Pages Error Messages ...................................................................... A-52
XML-25001: Cannot locate requested XSQL file. Check the name. .................... A-52
XML-25002: Cannot acquire database connection from pool: string .................. A-52
XML-25003: Failed to find config file string in CLASSPATH............................ A-53
XML-25004: Could not acquire a database connection named: string ............... A-53
XML-25005: XSQL page is not well-formed....................................................... A-53
XML-25006: XSLT stylesheets is not well-formed: string ................................. A-53
XML-25007: Cannot acquire a database connection to process page.................. A-53
XML-25008: Cannot find XSLT Stylesheet: string ........................................... A-53
XML-25009: Missing arguments on command line ............................................. A-53
XML-25010: Error creating: string
\nUsing standard output. .................................................................................. A-53
XML-25011: Error processing XSLT stylesheets: string ................................. A-53
XML-25012: Cannot read XSQL Page ............................................................... A-53
XML-25013: XSQL Page URI is null; check exact case of file name.................... A-53
XML-25014: Resulting page is an empty document or had multiple document elements..... A-53

XML-25016: Error parsing posted XML Document............................................. A-53
XML-25017: Unexpected Error Occurred ......................................................... A-53
XML-25018: Unexpected Error Occurred processing stylesheet string ................ A-54
XML-25019: Unexpected Error Occurred reading stylesheet string .................... A-54
XML-25020: Config file string is not well-formed. .............................................. A-54
XML-25021: Serializer string is not defined in XSQL configuration file .......... A-54
XML-25022: Cannot load serializer class string ............................................... A-54

xxxi
XML-25023: Class string is not an XSQL Serializer ......................................................... A-54
XML-25024: Attempted to get response Writer after getting OutputStream .................. A-54
XML-25025: Attempted to get response OutputStream after getting Writer ................. A-54
XML-25026: Stylesheet URL references an untrusted server ........................................ A-54
XML-25027: Failed to load string class for built-in xsq|:string action ............................ A-54
XML-25028: Error reading string. Check case of the name ............................................ A-54
XML-25029: Cannot load error handler class string ..................................................... A-54
XML-25030: Class string is not an XSQL ErrorHandler ............................................... A-54
XML-25100: You must supply a string attribute .......................................................... A-54
XML-25101: Fatal error in Stylesheet Pool ................................................................. A-54
XML-25102: Error instantiating class string ............................................................... A-54
XML-25103: Unable to load class string ...................................................................... A-55
XML-25104: Class string is not an XSQL ActionHandler ............................................. A-55
XML-25105: XML returned from PLSQL agent was not well-formed .............................. A-55
XML-25106: Invalid URL string ................................................................................ A-55
XML-25107: Error loading URL string ....................................................................... A-55
XML-25108: XML Document string is not well-formed .............................................. A-55
XML-25109: XML Document returned from database is not well-formed .................. A-55
XML-25110: XML Document in parameter string is not well-formed ........................ A-55
XML-25111: Problem including string ....................................................................... A-55
XML-25112: Error reading parameter value .................................................................. A-55
XML-25113: Error loading XSL transform string ....................................................... A-55
XML-25114: Parameter string has a null value ......................................................... A-55
XML-25115: No posted document to process ............................................................... A-55
XML-25116: No query statement supplied .............................................................. A-55
XML-25117: No PL/SQL function name supplied ....................................................... A-55
XML-25118: Stylesheet URL references an untrusted server ........................................ A-55
XML-25119: You must supply either the string or string attribute ......................... A-56
XML-25120: You selected fewer than the expected string values ............................. A-56
XML-25121: Cannot use 'xpath' to set multiple parameters ..................................... A-56
XML-25122: Query must be supplied to set multiple parameters ............................. A-56
XML-25123: Error reading string. Check case of the name ........................................ A-56
XML-25124: Error printing additional error information .......................................... A-56
XML-25125: Only one of (string) attributes is allowed ............................................. A-56
XML-25126: One of (string) attributes must be supplied ......................................... A-56

XML Pipeline Error Messages ......................................................................................... A-56
XML-30000: Error ignored in string: string .................................................................. A-56
XML-30001: Error occurred in execution of Process ..................................................... A-56
XML-30002: Only XML type(s) string allowed ............................................................ A-56
XML-30003: Error creating/writing to output string ...................................................... A-56
XML-30004: Error creating base url string ................................................................. A-56
XML-30005: Error reading input string ....................................................................... A-57
XML-30006: Error in processing pipedoc Error element ............................................. A-57
XML-30007: Error converting output to xml type required by dependent process .... A-57
XML-30008: A valid parameter target is required ....................................................... A-57
XML-30009: Error piping output to input ..................................................................... A-57
XML-30010: Process definition element string needs to be defined ........................ A-57
XML-30011: ContentHandler not available ................................................................. A-57
XML-30012: Pipeline components are not compatible ................................................... A-57
XML-30013: Process with output label string not found .............................................. A-57
XML-30014: Pipeline is not complete, missing output/outparam label called string.... A-57
XML-30016: Unable to instantiate class ............................................................................. A-57
XML-30017: Target is up-to-date, pipeline not executed .............................................. A-58

Java API for XML Binding (JAXB) Error Messages ............................................................ A-58
XML-32202: a problem was encountered because multiple <schemaBindings> were defined. A-58
XML-32203: a problem was encountered because multiple <class> name annotations were defined on node string. A-58
XML-32204: a problem was encountered because the name in <class> declaration contained a package name prefix string which was not allowed. A-58
XML-32205: a problem was encountered because the property customization was not specified correctly on node string. A-58
XML-32206: a problem was encountered because the javaType customization was not specified correctly on node string. A-58
XML-32207: a problem was encountered in declaring the baseType customization on the node string. A-58
XML-32208: a problem was encountered because multiple baseType customizations were declared on the node string. A-59
XML-32209: a problem was encountered because multiple javaType customizations were declared on the node string. A-59
XML-32210: a problem was encountered because invalid value was specified on customization of string. A-59
XML-32211: a problem was encountered because incorrect <schemaBindings> customization was specified. A-59
XML-32212: the <class> customization did not support specifying the implementation class using implClass declaration. The implClass declaration specified on node string was ignored. A-59
XML-32213: the <globalBindings> customization did not support specifying user specific class that implements java.util.List. The collectionType declaration was ignored. A-59

TXU Error Messages ......................................................................................................... A-59
General TXU Error Messages ............................................................................................ A-59
TXU-0001: Fatal Error ...................................................................................................... A-59
TXU-0002: Error ................................................................................................................ A-59
TXU-0003: Warning ......................................................................................................... A-60

DLF Error Messages ........................................................................................................ A-60
TXU-0100: parameter string in query string not found ................................................... A-60
TXU-0101: incompatible attributes col and constant coexist at string in query string ... A-60
TXU-0102: node string not found .................................................................................. A-60
TXU-0103: element string lacks content ........................................................................ A-60
TXU-0104: element string with SQL string lacks col or constant attribute............... A-60
TXU-0105: SQL exception string while processing SQL string ............................... A-60
TXU-0106: no data for column string selected by SQL string ................................. A-60
TXU-0107: datatype string not supported ................................................................. A-60
TXU-0108: missing maxsize attribute for column string ........................................ A-60
TXU-0109: a text length of string for string exceeds the allowed maximum of string.... A-61
TXU-0110: undeclared column string in row string .................................................. A-61
TXU-0111: lacking column data for string in row string ........................................... A-61
TXU-0112: undeclared query parameter string for column string .......................... A-61
TXU-0113: incompatible attribute string with a query on column string ............. A-61
TXU-0114: DLF parse error (string) on line string, character string in string .... A-61
TXU-0115: The specified date string string has an invalid format......................... A-61

TransX Informational Messages ............................................................................. A-61
TXU-0200: duplicate row at string ........................................................................ A-61

TransX Error Messages .......................................................................................... A-61
TXU-0300: document string not found .................................................................. A-61
TXU-0301: file string could not be read .............................................................. A-62
TXU-0302: archive string not found ....................................................................... A-62
TXU-0303: schema string not found in string ...................................................... A-62
TXU-0304: archive path for string not found ....................................................... A-62
TXU-0305: no database connection on string call for string .............................. A-62
TXU-0306: null tablename given; access denied ............................................... A-62
TXU-0307: lookup-keys could not be determined string ..................................... A-62
TXU-0308: binary file string not found ............................................................... A-62
TXU-0309: a file size of string exceeds the allowed maximum of 2,000 bytes ....... A-62

Assertion Messages ............................................................................................... A-62
TXU-0400: missing SQL statement element on string ........................................... A-62
TXU-0401: missing node string ........................................................................... A-62
TXU-0402: invalid flag string ............................................................................. A-63
TXU-0403: internal error string .......................................................................... A-63
TXU-0404: unexpected Exception string ............................................................ A-63

Usage Description Messages ............................................................................... A-63
TXU-0500: XML data transfer utility ................................................................. A-63
TXU-0501: Parameters are as follows: ................................................................. A-63
TXU-0502: JDBC connect string ......................................................................... A-63
TXU-0503: You can omit the connect string information through the '@' symbol  A-63
TXU-0504: Then jdbc:oracle:thin:@ will be supplied ......................................... A-63
TXU-0505: database username............................................................................ A-63
TXU-0506: database password .......................................................................... A-63
TXU-0507: data file name or URL ...................................................................... A-63
TXU-0508: Options: ............................................................................................ A-63
TXU-0509: update existing rows ......................................................................... A-64
TXU-0510: raise exception if a row is already existing ....................................... A-64
TXU-0511: print data in the predefined format .................................................... A-64
TXU-0512: save data in the predefined format .................................................... A-64
TXU-0513: print the XML to load ...................................................................... A-64
TXU-0514: print the tree for update ................................................................... A-64
TXU-0515: omit validation ................................................................................... A-64
TXU-0516: validate the data format and exit without loading .............................. A-64
TXU-0517: preserve whitespace ......................................................................... A-64
TXU-0518: Examples: ......................................................................................... A-64

XSU Error Messages ............................................................................................. A-64

Keywords and Syntax for Error Messages ........................................................... A-64
XSUK-0001: DOCUMENT ................................................................................... A-64
XSUK-0002: ROWSET

XSUK-0003: ROW

XSUK-0004: ERROR

XSUK-0005: num

XSUK-0006: item

Generic Error Messages

XSUE-0000: Internal Error -- Exception Caught

XSUE-0001: Internal Error --

XSUE-0002: string is not a scalar column. The row id attribute can only get values from scalar columns.

XSUE-0003: string is not a valid column name.

XSUE-0004: This object has been closed. If you would like the object not to be closed implicitly between calls, see the string method.

XSUE-0005: The row-set enclosing tag and the row enclosing tag are both omitted; consequently, the result can consist of at most one row which contains exactly one column which is not marked to be an XML attribute.

XSUE-0006: The row enclosing tag or the row-set enclosing tag is ommitted; consequently to get a well formed XML document, the result can only consist of a single row with multiple columns or multiple rows with exactly one column which is not marked to be an XML attribute.

XSUE-0007: Parsing of the sqlname failed -- invalid arguments.

XSUE-0008: Character string is not allowed in an XML tag name.

XSUE-0009: this method is not supported by string class. Please use string instead.

XSUE-0010: The number of bind names does not equal the number of bind values.

XSUE-0011: The number of bind values does not match the number of binds in the SQL statement.

XSUE-0012: Bind name identifier string does not exist in the sql query.

XSUE-0013: The bind identifier has to be of non-zero length.

XSUE-0014: Root node supplied is null.

XSUE-0015: Invalid LOB locator specified.

XSUE-0016: File string does not exit.

XSUE-0017: Can not create oracle.sql.STRUCT object of a type other than oracle.sql.STRUCT (i.e. ADT).

XSUE-0018: Null is not a valid DocumentHandler.

XSUE-0019: Null and empty string are not valid namespace aliases.

XSUE-0020: to use this method you will have to override it in your subclass.

XSUE-0021: You are using an old version of the gss library; thus, sql-xml name escaping is not supported.

XSUE-0022: cannot create XMLType object from opaque base type: string

Query Error Messages

XSUE-0100: Invalid context handle specified.

XSUE-0101: In the FIRST row of the resultset there is a nested cursor whose parent cursor is empty; when this condition occurs we are unable to generate a dtd.

XSUE-0102: string is not a valid IANA encoding.

XSUE-0103: The resultset is a "TYPE_FORWARD_ONLY" (non-scrollable); consequently, xsu can not reposition the read point. Furthermore, since the result set has been passed to the xsu by the caller, the xsu can not recreate the resultset.

XSUE-0104: input character is invalid for well-formed XML: string

DML Error Messages

...
XSUE-0200: The XML element tag *string* does not match the name of any of the columns/attributes of the target database object.  A-67
XSUE-0201: NULL is an invalid column name................................................................................. A-67
XSUE-0202: Column *string*, specified to be a key column, does not not exits in table *string*... A-67
XSUE-0203: Column *string*, specified as column to be updated, does not exist in the table *string*.  A-67
XSUE-0204: Invalid REF element - *string* - attribute *string* missing. .............................. A-67
XSUE-0206: Must specify key values before calling update routine. Use the *string* function. A-67
XSUE-0207: UpdateXML: No columns to update. The XML document must contain some non-key columns to update.  A-67
XSUE-0208: The key column array must be non empty.............................................................. A-68
XSUE-0209: The key column array must be non empty.............................................................. A-68
XSUE-0210: No rows to modify -- the row enclosing tag missing. Specify the correct row enclosing tag.  A-68
XSUE-0211: *string* encountered during processing ROW element *string* in the XML document.  A-68
XSUE-0212: *string* XML rows were successfully processed......................................................... A-68
XSUE-0213: All prior XML row changes were rolled back............................................................. A-68

Pieces of Error Messages ............................................................................................................. A-68
XSUE-0300: Note............................................................................................................................. A-68
XSUE-0301: Exception *string* caught: *string*............................................................................ A-68
XSUE-0302: column ...................................................................................................................... A-68
XSUE-0303: name.......................................................................................................................... A-68
XSUE-0303: invalid ....................................................................................................................... A-68
XSUE-0304: xml document............................................................................................................ A-68
XSUE-0305: template ..................................................................................................................... A-68

Glossary

Index
Preface

This Preface contains these topics:

- Documentation Accessibility
- Related Documents
- Conventions

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see these Oracle resources:

- Oracle XML DB Developer’s Guide
- Oracle Database XML C API Reference
- Oracle Database XML C++ API Reference
- Oracle Database XML Java API Reference
- Oracle Streams Advanced Queuing User’s Guide
- http://www.oracle.com/technology/tech/xml/

Many of the examples in this documentation are provided with your software in the following directories:

- $ORACLE_HOME/xdk/demo/java/
- $ORACLE_HOME/xdk/demo/c/
- $ORACLE_HOME/xdk/java/sample/
- $ORACLE_HOME/rdbms/demo
Many of the examples in this book use the sample schemas, which are installed by
default when you select the Basic Installation option with an Oracle Database
installation. Refer to Oracle Database Sample Schemas for information on how these
schemas were created and how you can use them yourself.

Printed documentation is available for sale in the Oracle Store at

http://oraclestore.oracle.com/

To download free release notes, installation documentation, white papers, or other
collateral, please visit the Oracle Technology Network (OTN). You must register online
before using OTN; registration is free and can be done at

http://www.oracle.com/technology/membership/

If you already have a username and password for OTN, then you can go directly to the
documentation section of the OTN Web site at

http://www.oracle.com/technology/documentation/

For additional information about XML, see:

- Oracle Database 10g XML & SQL: Design, Build, & Manage XML Applications in Java,
  C, C++, & PL/SQL by Mark Scardina, Ben Chang, and Jinyu Wang, Oracle Press,
  http://www.osborne.com/oracle/
- WROX publications, especially XML Design and Implementation by Paul Spencer,
  which covers XML, XSL, and development.
- Building Oracle XML Applications by Steve Muench, O'Reilly,
  http://www.oreilly.com/catalog/orxmlapp/
- The XML Bible, http://www.ibiblio.org/xml/books/biblegold/
- XML, Java, and the Future of the Web by Jon Bosak, Sun Microsystems,
  http://www.ibiblio.org/bosak/xml/why/xmlapps.htm
- XML for the Absolute Beginner by Mark Johnson, JavaWorld,
- XML And Databases by Ronald Bourret,
  http://www.rpbourret.com/xml/XMLAndDatabases.htm
- XML Specifications by the World Wide Web Consortium (W3C),
  http://www.w3.org/XML/
- XML.com, a broad collection of XML resources and commentary,
  http://www.xml.com/
- Annotated XML Specification by Tim Bray, XML.com,
  http://www.xml.com/axml/testaxml.htm
- XML.org, hosted by OASIS as a resource to developers of purpose-built XML
  languages, http://xml.org/

Conventions

The following text conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>boldface</td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td>Convention</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td>monospace</td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
What's New in the XDK?

What's New contains this topic:

- Features Introduced in Oracle XML Developer's Kit 11g Release 1 (11.1)

Features Introduced in Oracle XML Developer's Kit 11g Release 1 (11.1)

- DOM Stream access to XML nodes is done by PL/SQL and Java APIs. Nodes in an XML document can now exceed 64 KBytes by a large amount.

  See Also:  "Large Node Handling" on page 4-2

- The new XDK compact binary XML processors provide an encoder, a decoder, and a token manager to convert the schema or non-schema-based stream to and from XML 1.0 text and SAX events. The format is the same used in Oracle XML DB Developer's Guide.

  See Also:  Chapter 5, "Using Binary XML for Java"

- Binary XML adds a third storage model for persistent XML in the database: binary XML. Java and C APIs, and PL/SQL packages are affected. Binary XML support in the C API is used for both XML Developer's Kit and XML DB.

  See Also:  Chapter 19, "Using Binary XML for C"

Support is now provided for a scalable, pluggable DOM in Java:

- Scalable DOM support for Java includes lazy loading of DOM nodes, DOM updates of binary XML, multiple applications sharing the same DOM source, binary XML as output, and shadow copy in DOM.

- Pluggable DOM splits the DOM implementation into two separate layers: DOM API layer and data layer. The data is either internal or plug-in, and are accessed through an implementation of the InfosetReader interface.

- Configurable support for Java DOM is provided.

  See Also:  "Scalable DOM" on page 4-4

- The Unified Java API for XML allows mid-tier Java programs to leverage all the benefits of XMLType used with a session pool model of connection management. This allows XMLType object to be disconnected from the database session used to create it.
See Also: Chapter 2, "Unified Java API for XML"

- JAXP 1.3 support for XPath with extensions improves Java XSLT performance. Support is for static and dynamic context and users can register runtime context.
- HTTP server for SOA is enabled. The database is capable of exposing PL/SQL packages, procedures, and functions as Web Services. The database is capable of executing dynamic XQuery and SQL queries.

  See Also: Chapter 22, "Using SOAP with the C XDK"

- Chapters on SOAP for Java and C++ were removed.
- XmlDiff for C detects the differences between two XML documents and represents the difference in XML. XmlPatch outputs the differences and applies the changes on the target XML document.

  See Also: Chapter 21, "Determining XML Differences Using C"

- The JSR 170 standard is supported.

  See Also: "Standards and Specifications" on page 4-2

- The C Pull Parser reduces memory overhead compared to the SAX model.

  See Also: "What is the XML Pull Parser for C?" on page 18-12

- Streaming Validator support for C improves XML processing.

  See Also: "What is the Streaming Validator?" on page 20-4

- Document Updates For DLF/TransX is enhanced.

  See Also:
  - Chapter 12, "Using the TransX Utility"
  - Chapter 13, "Data Loading Format (DLF) Specification"
This chapter contains the following topics:

- Overview of Oracle XML Developer's Kit (XDK)
- XDK Components
- XML Document Generation with the XDK Components
- Development Tools and Frameworks for the XDK
- Installing the XDK

Overview of Oracle XML Developer's Kit (XDK)

Oracle Oracle XML Developer's Kit (XDK) is a versatile set of components that enables you to build and deploy C, C++, and Java software programs that process XML. You can assemble these components into an XML application that serves your business needs.

Note: Customers using Oracle XDK with PL/SQL and migrating from Oracle Database Release 8.1 or 9.2 are strongly encouraged to use AL32UTF8 as the database character set. Otherwise, issues can arise during PL/SQL processing of XML data that contains escaped entities.

Oracle XDK provides the foundation for the Oracle XML solution. The XDK supports Oracle XML DB, which is a set of technologies used for storage and processing of XML in the database. You can use the XDK in conjunction with Oracle XML DB to build applications that run in Oracle Database. You can also use the XDK independently of XML DB.

XML Date Formatting

Dates and timestamps in generated XML are now in the formats specified by XML Schema. See Oracle Oracle XML DB Developer’s Guide, Chapter "Generating XML Data from the Database", section "XMLELEMENT and XMLATTRIBUTES SQL Functions", subsection "Formatting of XML Dates and Timestamps".

The Oracle XDK is fully supported by Oracle and comes with a commercial redistribution license. The standard installation of Oracle Database includes the XDK.

Table 1–1 describes the XDK components, specifies which programming languages are supported, and directs you to sections that describe how to use the components.
## Table 1–1  Overview of XDK Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Lang.</th>
<th>Refer To</th>
</tr>
</thead>
</table>
| **XML Parser** | Creates and parses XML with industry standard DOM and SAX interfaces. | Java, C, C++ | - Chapter 4, "XML Parsing for Java"  
- Chapter 18, "Using the XML Parser for C"  
- Chapter 25, "Using the XML Parser for C++" |
| XML Compressor | Enables binary compression and decompression of XML documents. The compressor is built into the XML parser for Java. | Java | "Compressing XML" on page 4-42 |
| **Java API for XML Processing (JAXP)** | Enables you to use SAX, DOM, XML Schema processor, XSLT processors, or alternative processors, from your Java program. | Java | "Parsing XML with JAXP" on page 4-37 |
| XSLT Processor | Transforms XML into other text-based formats such as HTML. | Java, C, C++ | - Chapter 6, "Using the XSLT Processor for Java"  
- Chapter 17, "Using the XSLT and XVM Processors for C"  
- Chapter 26, "Using the XSLT Processor for C++" |
- Chapter 20, "Using the XML Schema Processor for C"  
- Chapter 27, "Using the XML Schema Processor for C++" |
| **XML Class Generator** | Generates Java or C++ classes from DTDs or XML schemas so that you can send XML data from Web forms or applications. The Java implementation supports Java Architecture for XML Binding (JAXB). | Java, C++ | Chapter 8, "Using the JAXB Class Generator" and Chapter 29, "Using the XML Class Generator for C++" |
| XML Pipeline Processor | Applies XML processes specified in a declarative XML Pipeline document. | Java | Chapter 9, "Using the XML Pipeline Processor for Java" |
| **XML JavaBeans** | Provides a set of bean encapsulations of XDK components for ease of use of Integrated Development Environment (IDE), Java Server Pages (JSP), and applets. | Java | Chapter 10, "Using XDK JavaBeans" |
| **XML SQL Utility (XSU)** | Generates XML documents, DTDs, and Schemas from SQL queries. Maps any SQL query result to XML and vice versa. The XSU Java classes are mirrored by PL/SQL packages. | Java, PL/SQL | Chapter 11, "Using the XML SQL Utility (XSU)" |
| TransX Utility | Loads translated seed data and messages into the database using XML. | Java | Chapter 12, "Using the TransX Utility" |
You can use the XDK components to perform various types of XML processing. For example, you can develop programs that do the following:

- Parse XML
- Validate XML against a DTD or XML schema
- Transform an XML document into another XML document by applying an XSLT stylesheet
- Generate Java and C++ classes from input XML schemas and DTDs

Figure 1–1 illustrates a hypothetical XML processor that performs the preceding tasks.

The XDK contains components that you can use in your programs. This section describes the following XDK components, some of which are illustrated in Figure 1–1:
XML Parsers

An XML parser is a processor that reads an XML document and determines the structure and properties of the data. It breaks the data into parts and provides them to other components.

An XML processor can programmatically access the parsed XML data with the following APIs:

- Use a SAX interface to serially access the data element by element. You can register event handlers with a SAX parser and invoke callback methods when certain events are encountered.
- Use DOM APIs to represent the XML document as an in-memory tree and manipulate or navigate it.

The XDK includes XML parsers for Java, C, and C++. Each parser includes support for both DOM and SAX APIs.

The XML parser for Java supports version 1.2 of JAXP, which is a standard API that enables use of DOM, SAX, XML Schema, and XSLT independently of a processor implementation. Thus, you can change the implementation of XML processors without impacting your programs.

The XML compressor is integrated into the XML parser for Java. It provides element-level XML compression and decompression with DOM and SAX interfaces. The compressor will compress XML documents without losing the structural and hierarchical information of the DOM tree. After parsing an XML document, you can serialize it with DOM or SAX to a binary stream and then reconstruct it later.

You can use the compressor to reduce the size of XML message payloads, thereby increasing throughput. When used within applications as the internal XML document access, it significantly reduces memory usage while maintaining fast access.

Figure 1–2 illustrates the functionality of the XDK parsers for Java, C, and C++.
XSLT Processors

eXtensible Stylesheet Language Transformation (XSLT) is a stylesheet language that enables processors to transform one XML document into another XML document. An XSLT document is a stylesheet that contains template rules that govern the transformation.

The Oracle XSLT processor fully supports the W3C XSL Transformations 1.0 recommendation. The processor also implements also implements the current working drafts of the XSLT and XPath 2.0 standards. It enables standards-based transformation of XML information inside and outside the database on any operating system.

The Oracle XML parsers include an integrated XSLT processor for transforming XML data by means of XSLT stylesheets. By using the XSLT processor, you can transform XML documents from XML to XML, to XHTML, or almost any other text format.

See Also:
- Chapter 4, "XML Parsing for Java"
- Chapter 18, "Using the XML Parser for C"
- Chapter 25, "Using the XML Parser for C++"

XML Schema Processors

The XML Schema language was created by the W3C to describe the content and structure of XML documents in XML, thus improving on DTDs. An XML schema contains rules that define validity for an XML application. Unlike a DTD, an XML schema is itself written in XML.

One of the principal advantages of an XML schema over a DTD is that a schema can specify rules for the content of elements and attributes. An XML schema specifies a set of built-in datatypes, for example, string, float, and date. Users can derive their own datatypes from the built-in datatypes. For example, the schema can restrict dates to those after the year 2000 or specify a list of legal values.

The Oracle XDK includes an XML Schema processor for Java, C, and C++. 

See Also:
- "Using the XSLT Processor for Java: Overview" on page 6-3.
- Specifications and other information are found on the W3C site at [http://www.w3.org/Style/XSL](http://www.w3.org/Style/XSL)
XML Class Generators

An XML class generator is a software program that accepts a parsed XML schema or DTD as input and generates Java or C++ source class files as output. The XDK includes both the JAXB class generator and the C++ class generator.

JAXB is a Java API and set of tools that map to and from XML data and Java objects. Because JAXB presents an XML document to a Java program in a Java format, you can write programs that process XML data without having to use a SAX parser or write callback methods. Each object derives from an instance of the schema component in the input document. JAXB does not directly support DTDs, but you can convert a DTD to an XML schema that is usable by JAXB. The XML class generator for C++ supports both DTDs and XML Schemas.

As an example of how to use JAXB, you can write a Java program that uses generated Java classes to build XML documents gradually. Suppose that you write an XML schema for use by a human resources department and a Java program that responds to users who change their personal data. The program can use JAXB to construct an XML confirmation document in a piecemeal fashion, which an XSLT processor can transform into XHTML and deliver to a browser.

See Also:

- Chapter 7, "Using the Schema Processor for Java"
- Chapter 20, "Using the XML Schema Processor for C"
- Chapter 27, "Using the XML Schema Processor for C++"

XML Pipeline Processor

The XML Pipeline Definition Language is an XML vocabulary for describing the processing relationships between XML resources. A document that is an instance of the pipeline language, that is, that defines the relationship between processes, is a pipeline document. For example, the document can specify that the program should first validate an input XML document and then, if it is valid, transform it.

See Also:

- Chapter 8, "Using the JAXB Class Generator"
- Chapter 29, "Using the XML Class Generator for C++"
Oracle XML Pipeline processor conforms to the XML Pipeline Definition Language 1.0 standard. The processor can take an input XML pipeline document and execute the pipeline processes according to the derived dependencies. The pipeline processor helps Java developers by replacing custom Java code with a simple declarative XML syntax for building XML processing applications.

See Also:  Chapter 9, "Using the XML Pipeline Processor for Java"

**XDK JavaBeans**

JavaBeans is a Java API for developing reusable software components that can be manipulated visually in a builder tool. A JavaBean is a Java object that conforms to this API. The Oracle XDK JavaBeans are a collection of visual and non-visual beans that are useful in a variety of XML-enabled Java programs or applets. Table 1–2 summarizes the XDK JavaBeans.

<table>
<thead>
<tr>
<th>JavaBean</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMBuilder</td>
<td>Builds a DOM Tree from an XML document. This bean is nonvisual.</td>
</tr>
<tr>
<td>XSLTransformer</td>
<td>Accepts an XML file, applies the transformation specified by an input XSLT stylesheet and creates the resulting output file. This bean is nonvisual.</td>
</tr>
<tr>
<td>DBAccess</td>
<td>Maintains CLOB tables that contain multiple XML and text documents.</td>
</tr>
<tr>
<td>XMLDBAccess</td>
<td>Extends the DBAccess bean to support the XMLType column, in which XML documents are stored in an Oracle Database table.</td>
</tr>
<tr>
<td>XMLDiff</td>
<td>Compares two XML DOM trees.</td>
</tr>
<tr>
<td>XMLCompress</td>
<td>Encapsulates the XML compression functionality.</td>
</tr>
<tr>
<td>XSDValidator</td>
<td>Encapsulates the oracle.xml.parser.schema.XSDValidator class and adds capabilities for validating a DOM tree.</td>
</tr>
</tbody>
</table>

See Also:  Chapter 10, "Using XDK JavaBeans"

**Oracle XML SQL Utility (XSU)**

XSU is a set of Java class libraries that you can use to perform the following tasks:

- Automatically and dynamically render the results of arbitrary SQL queries into canonical XML. XSU supports queries over richly-structured, user-defined object types and object views, including XMLType. When XSU transforms relational data into XML, the resulting XML document has the following structure:
  - Columns are mapped to top-level elements.
  - Scalar values are mapped to elements with text-only content.
  - Object types are mapped to elements with attributes appearing as sub-elements.
  - Collections are mapped to lists of elements.
- Load data from an XML document into an existing database schema or view.

Note: XSU also has a PL/SQL implementation. The DBMS_XMLQuery and DBMS_XMLSave PL/SQL packages reflect the functions in the OracleXMLQuery and OracleXMLSave Java classes.
Figure 1–4 illustrates how XSU processes SQL queries and returns the results as an XML document.

Figure 1–4  XSU Processes SQL Queries and Returns the Result as XML

Handling or Representing an XML Document
XSU can generate an XML document in any of the following ways:

■ A string representation of the XML document. Use this representation if you are returning the XML document to a requester.

■ An in-memory DOM tree. Use this representation if you are operating on the XML programmatically, for example, transforming it with the XSLT processor by using DOM methods to search or modify the XML.

■ A series of SAX events. You can use this functionality when retrieving XML, especially large documents or result sets.

Using XSU with an XML Class Generator
You can use XSU to generate an XML schema based on the relational schema of the underlying table or view that you are querying. You can use the generated XML schema as input to the JAXB class generator the C++ class generator. You can then write code that uses the generated classes to create the infrastructure behind a Web-based form. Based on this infrastructure, the form can capture user data and create an XML document compatible with the database schema. A program can write the XML directly to the corresponding table or object view without further processing.

See Also:  Chapter 11, "Using the XML SQL Utility (XSU)"

TransX Utility
The Oracle TransX utility is a data transfer utility that enables you to populate a database with multilingual XML data. It uses a simple data format that is intuitive for both developers and translators and uses a validation capability that is less error-prone than previous techniques.

How is the TransX utility different from XSU? TransX utility is an application of XSU that loads translated seed data and messages into a database schema. If you have data to be populated into a database in multiple languages, then the utility provides the functionality that you would otherwise need to develop with XSU.
XSQL Pages Publishing Framework

The XSQL pages publishing framework (XSQL servlet) is a server component that processes an XSQL file, which is an XML file with a specific structure and grammar, and produces dynamic XML documents from one or more SQL queries of data objects. Figure 1–5 shows you can invoke the XSQL servlet.

![Figure 1–5 XSQL Pages Publishing Framework](image)

The XSQL servlet uses the Oracle XML parser to process the XSQL file, passing XSLT processing statements to its internal processor while passing parameters and SQL statements between the tags to XSU. Results from those queries are received as XML-formatted text or a JDBC `ResultSet` object. If necessary, you can further transform the query results by using the built-in XSLT processor.

One example of an XSQL servlet is a page that contains a query of flight schedules for an airline with a bind variable for the airport name. The user can pass an airport name as a parameter in a web form. The servlet binds the parameter value in its database query and transforms the output XML into HTML for display in a browser.

See Also: Chapter 14, "Using the XSQL Pages Publishing Framework"

Soap Services

Simple Object Access Protocol (SOAP) is a platform-independent messaging protocol that enables programs to access services, objects, and servers. Oracle SOAP Services is published and executed through the Web and provides the standard XML message format for all programs. With SOAP Services, you can use the XDK to develop messaging, RPC, and Web service programs with XML standards.

See Also: Chapter 22, "Using SOAP with the C XDK"

XSLT Virtual Machine (XVM)

The XVM for C/C++ is the software implementation of a CPU designed to run compiled XSLT code. To run this code, you need to compile XSLT stylesheets into byte
code that the XVM engine understands. Figure 1–6 illustrates how the XVM processes XML and XSL.

**Figure 1–6  XSLT Virtual Machine**

The XDK includes an XSLT compiler that is compliant with the XSLT 1.0 standard. The compilation can occur at runtime or be stored for runtime retrieval. Applications perform transformations more quickly with higher throughput because the stylesheet does not need to be parsed and the templates are applied based on an index lookup instead of an XML operation.

**See Also:** "XVM Processor" on page 17-1

**XML Document Generation with the XDK Components**

The XDK enables you to map the structure of an XML document to a relational schema. You can use the XDK to establish a two-way path to an Oracle database in which your program creates XML documents from tables and inserts XML-tagged data into tables. Each XDK programming language supports the development of programs that generate XML documents from relational data.

This section contains the following topics:

- XML Document Generation with Java
- XML Document Generation with C
- XML Document Generation with C++

**XML Document Generation with Java**

As shown in Figure 1–7, you can execute a SQL query against the database in three different ways. Table 1–3 describes the alternatives.

**Table 1–3  Generating XML in Java**

<table>
<thead>
<tr>
<th>Technology</th>
<th>Label in Figure 1–7</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSQL Servlet</td>
<td>A</td>
<td>Includes XSU and the XML parser</td>
</tr>
<tr>
<td>XSU</td>
<td>B</td>
<td>Includes XML parser</td>
</tr>
<tr>
<td>JDBC</td>
<td>C</td>
<td>Sends output data to the XML parser</td>
</tr>
</tbody>
</table>
Regardless of how your software program generates the XML from the database, Figure 1–7 illustrates possible further processing that your program can perform on the XML document. Table 1–4 describes some of the components that you can use to perform this additional processing.

**Table 1–4 Additional Document Processing with the Java XDK**

<table>
<thead>
<tr>
<th>Technology</th>
<th>Label in Figure 1–7</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAXB</td>
<td>D</td>
<td>Generates Java class files that correspond to an input XML Schema</td>
</tr>
<tr>
<td>JavaBeans</td>
<td>E</td>
<td>Can compare an XML document with another XML document</td>
</tr>
<tr>
<td>XSLT</td>
<td>F</td>
<td>Transforms the XML document into XHTML with an XSLT stylesheet</td>
</tr>
</tbody>
</table>
XML Document Generation with C

Figure 1–8 illustrates the Oracle XDK C language components that you can use to generate XML documents from relational data. The XDK C components are listed in Table 1–1.

Figure 1–8 Generating XML Documents with XDK C Components

As illustrated in Figure 1–8, you can use the XDK to develop a C program that processes an XML document as follows:

1. Send SQL queries to the database by the Oracle Call Interface (OCI) or the Pro*C/C++ Precompiler. The program must leverage the XML DB XML view functionality.
2. Process the resulting XML data with the XML parser or from the CLOB as an XML document.
3. Transform the document with the XSLT processor, send it to an XML-enabled browser, or send it for further processing to a software program.

XML Document Generation with C++

Figure 1–9 shows the Oracle XDK C++ components that you can use to generate XML documents. The XDK C++ components are listed in Table 1–1.
As illustrated in Figure 1–9, you can use the XDK to develop a C++ program that processes an XML document as follows:

1. Send SQL queries to the database by the Oracle C++ Call Interface (OCCI) or the Pro*C/C++ Precompiler.

2. Process the resulting XML data with the XML parser or from the CLOB as an XML document.

3. Transform the document with the XSLT processor, send it to an XML-enabled browser, or send it for further processing to a software program.

**Development Tools and Frameworks for the XDK**

Figure 1–10 illustrates some of the tools and frameworks that you can use to develop software programs that use XDK components. For example, you can use Oracle JDeveloper to write a Java client that can query the database, generate XML, and perform additional processing. An employee can then use this program to send a query to an Oracle database. The program can transfer XML documents to XML-based
business solutions for data exchange with other users, content and data management, and so forth.

Figure 1–10  XDK Tools and Frameworks

This section describes some of the tools and frameworks that you can use in e-business development:

- **Oracle JDeveloper**
- **User Interface XML (UIX)**
- **Oracle Reports**
- **Oracle XML Gateway**
- **Oracle Data Provider for .NET**

**Oracle JDeveloper**

Oracle JDeveloper is a J2EE development environment with end-to-end support for developing, debugging, and deploying e-business applications. JDeveloper provides a comprehensive set of integrated tools that support the complete development life
Development Tools and Frameworks for the XDK

Introduction to Oracle XML Developer's Kit

The development cycle, from source code control, modeling, and coding through debugging, testing, profiling, and deployment, JDeveloper simplifies development by providing deployment tools to create J2EE components such as the following:

- Applets
- JavaBeans
- Java Server Pages (JSP)
- Servlets
- Enterprise JavaBeans (EJB)

JDeveloper also provides a public API to extend and customize the development environment and integrate it with external products.

The Oracle XDK is integrated into JDeveloper, offering many ways to manage XML. For example, you can use the XSQL Servlet to perform the following tasks:

- Query and manipulate database information
- Generate XML documents
- Transform XML with XSLT stylesheets
- Deliver XML on the Web

JDeveloper has an integrated XML schema-driven code editor for working on XML Schema-based documents such as XML schemas and XSLT stylesheets. By specifying the schema for a certain language, the editor can assist you in creating a document in that markup language. You can use the Code Insight feature to provide a list of valid alternatives for XML elements or attributes in the document.

Oracle JDeveloper simplifies the task of working with Java application code and XML data and documents at the same time. It features drag-and-drop XML development modules such as the following:

- Color-coded syntax highlighting for XML
- Built-in syntax checking for XML and XSL
- Editing support for XML schema documents
- XSQL Pages and Servlet support
- Oracle's XML parser for Java
- XSLT processor
- XDK for JavaBeans components
- XSQL Page Wizard
- XSQL Action Handlers
- Schema-driven XML editor

See Also:

- [http://www.oracle.com/technology/forums](http://www.oracle.com/technology/forums) for the online discussion forum for JDeveloper
User Interface XML (UIX)

UIX (User Interface XML) is a framework for developing XML-enabled Web applications. The main focus of UIX is the user presentation layer of a program, with additional functionality for managing events and application flow. You can use UIX to create programs with page-based navigation, such as an online human resources program, rather than full-featured programs requiring advanced interaction, such as an integrated development environment (IDE).

See Also:
- [http://www.oracle.com/technology/sample_code/products/jdev/content.html](http://www.oracle.com/technology/sample_code/products/jdev/content.html) for sample JDeveloper Demonstration code for UIX
- JDeveloper online help for the complete UIX Developer’s Guide

Oracle Reports

Oracle Reports Developer and Reports Server is a development tool that enables you to build and publish dynamically generated Web reports. A wizard expedites the use of each major task. Report templates and live data previews allow you to customize the report structure. You can publish reports throughout the enterprise through a standard Web browser in formats such as the following:

- XML
- HTML with or without CSS
- PDF
- Text
- RTF
- PostScript
- PCL

See Also:
- [http://www.oracle.com/technology/products/reports](http://www.oracle.com/technology/products/reports) for links to Oracle Reports documentation

Oracle XML Gateway

Oracle XML Gateway is a set of services that enables integration with the Oracle E-Business Suite to create and consume XML messages triggered by business events. It integrates with Oracle Streams Advanced Queuing to enqueue and dequeue a message, which it can then transmit to or from the business partner through any message transport agent.

See Also:
- Oracle Streams Advanced Queuing User’s Guide
- Oracle XML DB Developer’s Guide

Oracle Data Provider for .NET

Oracle Data Provider for .NET (ODP.NET) is an implementation of a data provider for the Oracle Database. ODP.NET uses Oracle native APIs to offer fast and reliable access
Installing the XDK

Introduction to Oracle XML Developer’s Kit

1-17

to Oracle data and features from any .NET application and also uses and inherits classes and interfaces available in the Microsoft .NET Framework Class Library.

You can use ODP.NET and the XDK to extract data from relational and object-relational tables and views as XML documents. The use of XML documents for insert, update, and delete operations to the database server is also allowed. ODP.NET supports XML natively in the database through XML DB.

ODP.NET supports XML with features that:

■ Store XML data natively in the database server as the Oracle native type XMLType.

■ Access relational and object-relational data as XML data from an Oracle Database instance into Microsoft .NET environment and process the XML with the Microsoft .NET framework.

■ Save changes to the database server with XML data.

For the .NET application developer, features include the following:

■ Enhancements to the OracleCommand, OracleConnection, and OracleDataReader classes

■ XML-specific classes:
  – OracleXmlType
  – OracleXmlStream
  – OracleXmlQueryProperties
  – OracleXmlSaveProperties

See Also: Oracle Data Provider for .NET Developer’s Guide

Installing the XDK

This section assumes that you installed Oracle Database from either CD-ROM or from an archive downloaded from Oracle Technology Network (OTN). The Oracle Database CD installs the Oracle XDK by default. Note that you must install the demo programs from the Oracle Database Examples media to obtain the XDK demos. This manual presumes that you have access to the XDK demos programs.

After installing Oracle Database and the demos from the Oracle database Examples media, your Oracle Database home should be set up as follows:

- Oracle_home_directory
  | - bin: includes XDK executables
  | - lib: includes XDK libraries
  | - jlib: includes Globalization Support libraries for the XDK
  | - nls: includes binary files used as part of globalization support
  | - xdk: XDK scripts, message files, documentation, and demos
    readme.html
    | - admin: SQL scripts and XSL Servlet Configuration
      file (XSQLConfig.xml)
    | - demo: sample programs (installed from Oracle Database Examples media)
      | - c
      | - cpp
      | - java
      | - jsp
    | - doc: release notes and readme
      content.html
      index.html
      license.html
title.html
| - cpp
| - images
| - java
| - include: header files
| - mesg: error message files

The directory that contains the XDK is called the **XDK home**. Set the `$XDK_HOME` environment variable (UNIX) or the `%XDK_HOME%` variable (Windows) to the XDK directory in your Oracle home. For example, you can set use `csh` on UNIX to set the XDK home as follows:

```
setenv XDK_HOME $ORACLE_HOME/xdk
```

**See Also:**
- Chapter 3, "Getting Started with Java XDK Components"
- Chapter 16, "Getting Started with C XDK Components"
- Chapter 23, "Getting Started with C++ XDK Components"
This part contains chapters describing how to use Oracle XDK in Java development.

This part contains the following chapters:

- Chapter 3, "Getting Started with Java XDK Components"
- Chapter 2, "Unified Java API for XML"
- Chapter 4, "XML Parsing for Java"
- Chapter 5, "Using Binary XML for Java"
- Chapter 6, "Using the XSLT Processor for Java"
- Chapter 7, "Using the Schema Processor for Java"
- Chapter 8, "Using the JAXB Class Generator"
- Chapter 9, "Using the XML Pipeline Processor for Java"
- Chapter 10, "Using XDK JavaBeans"
- Chapter 11, "Using the XML SQL Utility (XSU)"
- Chapter 12, "Using the TransX Utility"
- Chapter 13, "Data Loading Format (DLF) Specification"
- Chapter 14, "Using the XSQL Pages Publishing Framework"
- Chapter 15, "Using the XSQL Pages Publishing Framework: Advanced Topics"
This chapter introduces you to the Unified Java API for XMLType and provides information about the APIs that are unified for Oracle XML DB and Oracle XML Developer’s Kit.

This chapter contains these topics:

- **Overview of Unified Java API**
- **Component Unification**
- **Moving to the Unified Java API Model**

### Overview of Unified Java API

Unified Java API is a programming interface that combines the functionality required by both Oracle XDK and Oracle XML DB. Oracle XML DB implements Java DOM API using the Java package `oraclexdb.dom` and Oracle XML Developer’s Kit implements it using the `oracle.xml.parser.v2` package. With Unified Java APIs, you can use a unified set of core DOM APIs required by both Oracle XDK and Oracle XML DB, as well as make use of the new Java classes that provide extra functionality that is built on top of the DOM API.

You can use Unified Java APIs irrespective of how your XML data is stored, that is, whether it resides within or outside the database. This is because Unified Java APIs use a session pool model of connection management. If you do not specify the connection type as **thick** (that uses OCI APIs and is C-based) or **thin** (that uses JDBC APIs and is pure Java-based), then Java Document Object Model (DOM) APIs are used for connecting to a local document object that is not stored in the database.

**See Also:** Oracle Database XML Java API Reference for information about the `oracle.xml.parser.v2` package.

### Component Unification

Certain components that were either supported by only the **thick** connection or the **thin** connection have been unified in the Unified Java API model. The components that were earlier supported by **thin** connection but have been unified include:

- DOM Parser
- JAXP Transformer
- XSU
- XSLT
Moving to the Unified Java API Model

Unified Java API provides new Java classes that replace the old oracle.xdb.dom Java classes. All classes in the oracle.xdb.dom package have been deprecated. If you are using any of the old classes, you need to migrate to the new Unified Java API and use the oracle.xml.parser.v2 classes instead.

Java DOM APIs for XMLType Classes

Table 2–1 lists the oracle.xdb.dom package classes that have been deprecated in the Oracle Database 11g release 1.

<table>
<thead>
<tr>
<th>oracle.xdb.dom.* classes</th>
<th>oracle.xml.parser.v2 (Unified Java API) classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDBAttribute</td>
<td>XMLAttr</td>
</tr>
<tr>
<td>XDBBinaryDocument</td>
<td>This has been deprecated and has no replacement in the Java DOM API XMLType classes</td>
</tr>
<tr>
<td>XDBCData</td>
<td>XMLCDATA</td>
</tr>
<tr>
<td>XDBCCharData</td>
<td>CharData</td>
</tr>
<tr>
<td>XDBComment</td>
<td>XMLComment</td>
</tr>
<tr>
<td>XDBDocFragment</td>
<td>XMLDocumentFragment</td>
</tr>
<tr>
<td>XDBDocument</td>
<td>XMLDocument</td>
</tr>
<tr>
<td>XDBDocumentType</td>
<td>DTD</td>
</tr>
<tr>
<td>XDBDOMException</td>
<td>XMLDOMException</td>
</tr>
<tr>
<td>XDBDomImplementation</td>
<td>XMLDomImplementation</td>
</tr>
<tr>
<td>XDBDONotFoundErrException</td>
<td>This has been deprecated and has no replacement in the Java DOM API XMLType classes</td>
</tr>
<tr>
<td>XDBElement</td>
<td>XMLElement</td>
</tr>
<tr>
<td>XDBEntity</td>
<td>XMLEntity</td>
</tr>
<tr>
<td>XDBEntityReference</td>
<td>XMLEntityReference</td>
</tr>
<tr>
<td>XDBNamedNodeMap</td>
<td>XMLAttrList</td>
</tr>
<tr>
<td>XDBNode</td>
<td>XMLNode</td>
</tr>
<tr>
<td>XDBNotation</td>
<td>XMLNotation</td>
</tr>
<tr>
<td>XDBNotImplementedException</td>
<td>This has been deprecated and has no replacement in the Java DOM API XMLType classes</td>
</tr>
<tr>
<td>XDBProcInst</td>
<td>XMLPI</td>
</tr>
<tr>
<td>XDBText</td>
<td>XMLText</td>
</tr>
</tbody>
</table>

When you use the Java DOM API to retrieve XML data, you either get an XMLDocument instance if the connection is thin, or an XDBDocument instance with method getDOM() and an XMLDocument instance with method getDocument(). Both XMLDocument and XDBDocument are instances of the W3C DOM interface. The getDOM() method and XDBDocument class have been deprecated in the Unified Java APIs. Table 2–2 lists the new methods and classes that are supported with the current release.
Extension APIs

In addition to the W3C Recommendation, the Unified Java API implementation provides some extension APIs that extend the W3C DOM APIs in various ways. You can use the Oracle-specific extension APIs for either performing the basic functions like connecting to a database or for performance enhancement.

`XMLDocument` is a class that represents the DOM for the instantiated XML document. You can retrieve the `XMLType` value from the XML document using the `XMLType` constructor that takes a `Document` argument:

```java
XMLType createXML(Connection conn, Document domdoc)
```

Use the `freeNode()` extension API available in the `oracle.xml.parser.v2` package (`XMLNode` class) for manual dereferencing of nodes. When you use `freeNode()`, you explicitly dereference a document fragment from the DOM tree.

Document Creation Java APIs

The unified Java APIs that create an XMLDocument must create either a thin document or a thick document. A thick document requires a `Connection` object in order to establish communication with the database. So the creation APIs are extended to accept a `Connection` object.

| `XMLType.createXML` APIs - Based on connection type. That is, Thick DOM for OCI or KPRB connection, and Thin DOM for a Thin connection. |

| Table 2–3  XMLDocument Output Based on XMLDocument.KIND and XMLDocument.CONNECTION |
|-----------------------------|-----------------------------|-----------------------------|
| `XMLDocument.KIND` | `XMLDocument.CONNECTION` | `XMLDocument` |
| `XMLDocument.THICK` | Thick or KPRB connection | Thick DOM |
| `XMLDocument.THICK` | Thin or no connection | Exception |
| `XMLDocument.THIN` | Any connection type | Thin DOM |
| Not specified | Any connection type | Non-XMLType APIs. Thin DOM XMLType.createXML APIs - Based on connection type. That is, Thick DOM for OCI or KPRB connection, and Thin DOM for a Thin connection. |

The following objects and methods are provided for document creation in the unified Java API model:
- **DOMParser object and parse() method**: Use the DOMParser object and parse() method to parse XML documents. You must specify the type of object, that is, thick or thin. For thick objects, you must also provide the Connection property using the DOMParser.setAttribute() API. For example:

```java
DOMParser parser = new oracle.xml.parser.v2.DOMParser();
parser.setAttribute(XMLDocument.KIND, XMLDocument.THICK);
parser.setAttribute(XMLDocument.CONNECTION, conn);
```

- **DocumentBuilder object**: Use the DocumentBuilder object to parse XML documents using the Java-specific API, JAXP. You need to create a DOM parser factory with the DocumentBuilderFactory class. The DocumentBuilderFactory class then passes the connection into the DOMParser, which is used to create the document from these APIs.

- **DocumentBuilder builds DOM from input SAX events**: This takes the Connection from a property set on the DocumentBuilderFactory. For example:

```java
DocumentBuilderFactory.setAttribute(XMLDocument.CONNECTION, conn);
DocumentBuilderFactory.setAttribute(XMLDocument.KIND, XMLDocument.THICK);

DocumentBuilderFactory passes the connection into the DOMParser that is used to create the document from the following APIs:

```java
DocumentBuilder.newDocument()
DocumentBuilder.parse(InputStream)
DocumentBuilder.parse(InputStream, int)
DocumentBuilder.parse(InputSource)
```

- **XSU**: These methods return an XMLDocument to the user. You can specify whether they want a thick or thin object:

```java
OracleXMLUtil util = new OracleXMLUtil(...);
util.setAttribute(XMLDocument.KIND, XMLDocument.THICK);
util.setAttribute(XMLDocument.CONNECTION, conn);
Document doc = util.getXMLDOMFromStruct(struct, enc);

OracleXMLQuery query = new OracleXMLQuery(...);
query.setAttribute(XMLDocument.KIND, XMLDocument.THICK);
query.setAttribute(XMLDocument.CONNECTION, conn);
Document doc = query.getXMLDOM(root, meta);

OracleXMLDocGenDOM dgd = new OracleXMLDocGenDOM(...);
dgd.setAttribute(XMLDocument.KIND, XMLDocument.THICK);
dgd.setAttribute(XMLDocument.CONNECTION, conn);
Document doc = dgd.getXMLDocumentDOM(struct, enc);
```

- **XMLType**: Using the getDocument() method, you can specify whether you want a thick or thin object. All existing XMLType methods are still available. In this example, the connection is inferred from the OPAQUE:

```java
XMLType xt = XMLType.createXML(orset.getOPAQUE(1), XMLDocument.THICK);
Document doc = xt.getDocument();
```

One known case that will not allow for the user to specify the type is the case of an XMLType that is created using the ResultSet.getObject() API. If a user has a table with an XMLType column, and the user selects this column, the user can call getObject() on the ResultSet. This will return an XMLType object. The type is determined by the Connection used in the JDBC call to fetch the column.
This chapter contains these topics:

- Installing Java XDK Components
- Java XDK Component Dependencies
- Setting Java XDK Environment Variables for UNIX
- Setting Java XDK Environment Variables for Windows
- Verifying the Java XDK Components Version

## Installing Java XDK Components

The Java XDK components are included with Oracle Database. This chapter assumes that you have installed XDK with Oracle Database and also installed the demo programs on the Oracle Database Examples media. Refer to "Installing the XDK" on page 1-17 for installation instructions and a description of the XDK directory structure.

Example 3–1 shows the UNIX directory structure for the XDK demos and the libraries used by the XDK components. The $ORACLE_HOME/xdk/demo/java subdirectories contain sample programs and data files for the XDK for Java components. The chapters in Part I, "XDK for Java" explain how to understand and use these programs.

### Example 3–1  Java XDK Libraries, Utilities, and Demos

- Oracle_home_directory
  - bin/
    - orajaxb
    - orapipe
    - oraxml
    - oraxsl
    - transx
  - lib/
    - classgen.jar
    - jdev-rt.zip
    - oraclesql.jar
    - transx.zip
    - xml.jar
    - xml2.jar
    - xmlcomp.jar
    - xmlcomp2.jar
    - xmldemo.jar
    - xmlmsg.jar
    - xmparserv2.jar
    - xschema.jar
Java XDK Component Dependencies

The subdirectories contain sample programs and data files for the Java XDK components. The chapters in Part I, "XDK for Java" explain how to use these programs to gain an understanding of the most important Java features.

Java XDK Component Dependencies

The Java XDK components are certified and supported with JDK version 5 and version 6. Earlier versions of Java are no longer supported. Figure 3–1 shows the dependencies of Java XDK components when using JDK 5.

Figure 3–1 Java XDK Component Dependencies for JDK 5

<table>
<thead>
<tr>
<th>TransX Utility (xml.jar)</th>
<th>JavaBeans (xmldemo.jar, xml.jar)</th>
<th>XSQL Servlet (xml.jar)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Generator (xsuil2.jar, xdb.jar)</td>
<td>JDBC Driver (ojdbc5.jar)</td>
<td>XML SQL Utility (xml.jar)</td>
</tr>
<tr>
<td>JDK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Java XDK components require the libraries alphabetically listed in Table 3–1. Note that some of the libraries are not specific to the XDK, but are shared among other Oracle Database components.
## Table 3–1  Java Libraries for XDK Components

<table>
<thead>
<tr>
<th>Library</th>
<th>Directory</th>
<th>Includes . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>classgen.jar</td>
<td>$ORACLE_HOME/lib</td>
<td>XML class generator for Java runtime classes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This library is maintained for backward compatibility only. You should use the JAXB class generator in xml.jar instead.</td>
</tr>
<tr>
<td>jdev-rt.zip</td>
<td>$ORACLE_HOME/lib</td>
<td>Java GUI libraries for use when working with the demos with the JDeveloper IDE.</td>
</tr>
<tr>
<td>ojdbc5.jar, ojdbc6.jar</td>
<td>$ORACLE_HOME/jdbc/lib</td>
<td>Oracle JDBC drivers for Java 5, 6. This JAR depends on orai18n.jar for character set support if you use a multibyte character set other than UTF-8, ISO8859-1, or JA16SJIS.</td>
</tr>
<tr>
<td>oraclesql.jar</td>
<td>$ORACLE_HOME/lib</td>
<td>Most of the XSQL Servlet classes needed to construct XSQL pages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This archive is superseded by xml.jar and is maintained for backward compatibility only.</td>
</tr>
<tr>
<td>orai18n.jar</td>
<td>$ORACLE_HOME/jlib</td>
<td>Globalization support for JDK 1.2 or above. It is a wrapper of all other Globalization jars and includes character set converters. If you use a multibyte character set other than UTF-8, ISO8859-1, or JA16SJIS, then place this archive in your CLASSPATH so that JDBC can convert the character set of the input file to the database character set when loading XML files with XSU, TransX Utility, or XSQL Servlet.</td>
</tr>
<tr>
<td>orai18n-collation.jar</td>
<td>$ORACLE_HOME/jlib</td>
<td>Globalization collation features: the OraCollator class and the lx3*.glb and lx4001[0-9].glb files.</td>
</tr>
<tr>
<td>orai18n-mapping.jar</td>
<td>$ORACLE_HOME/jlib</td>
<td>Globalization locale and character set name mappings: the OraResourceBundle class and lx4000[0-9].glb files. This archive is mainly used by the products that need only locale name mapping tables.</td>
</tr>
<tr>
<td>orai18n-utility.jar</td>
<td>$ORACLE_HOME/jlib</td>
<td>Globalization locale objects: the OraLocaleInfo class, the OraNumberFormat and OraDateFormat classes, and the lx[01]*.glb files.</td>
</tr>
<tr>
<td>transx.zip</td>
<td>$ORACLE_HOME/lib</td>
<td>TransX Utility classes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This archive is replaced by xml.jar and is retained for backward compatibility only.</td>
</tr>
<tr>
<td>xdb.jar</td>
<td>$ORACLE_HOME/rdbms/jlib</td>
<td>Classes needed by xml.jar and xmlcomp2.jar to access XMLType. It also includes classes needed to access the XML DB Repository as well as the XMLType DOM classes for manipulation of the DOM tree.</td>
</tr>
<tr>
<td>xml.jar</td>
<td>$ORACLE_HOME/lib</td>
<td>Classes from the following libraries:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- oraclesql.jar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- xsqlserializers.jar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- xmlcomp.jar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- xmlcomp2.jar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- transx.jar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The archive also contains the JAXB and Pipeline Processor classes.</td>
</tr>
</tbody>
</table>
Java XDK Component Dependencies

<table>
<thead>
<tr>
<th>Library</th>
<th>Directory</th>
<th>Includes ...</th>
</tr>
</thead>
</table>
| xmlcomp.jar     | $ORACLE_HOME/lib  | XML JavaBeans that do not depend on the database: DOMBuilder, XSLTransformer, DBAccess, XSDValidator, and XMLDiffer.          
|                 |                   | **Note:** This archive is included for backward compatibility only because its classes are included in xml.jar. They do not include the visuals Beans included in previous releases. |
| xmlcomp2.jar    | $ORACLE_HOME/lib  | XML JavaBeans that depend on the database: XMLDBAccess and XMLCompress. Thus, it depends on xdb.jar, which includes the classes that support XML DB. |
|                 |                   | **Note:** This JAR is included for backward compatibility only because its classes are included in xml.jar. They do not include the visuals Beans included in previous releases. |
| xmldemo.jar     | $ORACLE_HOME/lib  | The visual JavaBeans: XMLTreeView, XMLTransformPanel, XMLSourceView, and DBViewer. |
| xmlmsg.jar      | $ORACLE_HOME/lib  | Needed if you use XML parser with a language other than English. |
| xmlparserv2.jar | $ORACLE_HOME/lib  | APIs for the following:                                                      |
|                 |                   | ■ DOM and SAX parsers                                                        |
|                 |                   | ■ XML Schema processor                                                       |
|                 |                   | ■ XSLT processor                                                             |
|                 |                   | ■ XML compression                                                            |
|                 |                   | ■ JAXP                                                                       |
|                 |                   | ■ Utility functionality such as XMLSAXSerializer and asynchronous DOM Builder |
|                 |                   | This library includes xschema.jar.                                           |
| xschema.jar     | $ORACLE_HOME/lib  | Includes the XML Schema classes contained in xmlparserv2.jar.                |
|                 |                   | **Note:** This JAR file is maintained for backward compatibility only.       |
| xsqlserializers.jar | $ORACLE_HOME/lib | Serializer classes for XSQL Servlet needed for serialized output such as PDF. |
|                 |                   | **Note:** This archive is superseded by xml.jar and is maintained for backward compatibility only. |
| xsu12.jar       | $ORACLE_HOME/lib  | Classes that implement XSU. These classes have a dependency on xdb.jar for XMLType access. |

See Also:

- *Oracle Database Globalization Support Guide* to learn about the Globalization Support libraries
- *Oracle Database JDBC Developer’s Guide and Reference* to learn about the JDBC libraries
- *Oracle XML DB Developer’s Guide* to learn about XML DB
Setting Up the Java XDK Environment

In the Oracle Database installation of the XDK, you must manually set the $CLASSPATH (UNIX) or %CLASSPATH% (Windows) environment variables. Alternatively, set the -classpath option when compiling and running Java programs at the command line.

This section contains the following topics:

- Setting Java XDK Environment Variables for UNIX
- Setting Java XDK Environment Variables for Windows

Setting Java XDK Environment Variables for UNIX

Table 3–2 describes the UNIX environment variables required for use with the Java XDK components.

Table 3–2  UNIX Environment Settings for Java XDK Components

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
</table>
| $CLASSPATH    | Includes the following (note that a single period "." to represent the current directory is not required but may be useful):
|               | .:${CLASSPATHJ}:${ORACLE_HOME}/lib/xmlparserv2.jar:
|               | ${ORACLE_HOME}/lib/xsu12.jar:${ORACLE_HOME}/lib/xml.jar |
| $CLASSPATHJ   | For JDK 5, set as follows:
|               | CLASSPATHJ=${ORACLE_HOME}/jdbc/lib/ojdbc5.jar:${ORACLE_HOME}/jlib/orai18n.jar |
| $JAVA_HOME    | Installation directory for the Java JDK, Standard Edition. Modify the path that links to the Java SDK. |
| $LD_LIBRARY_PATH | For OCI JDBC connections:
|               | ${ORACLE_HOME}/lib:${LD_LIBRARY_PATH} |
| $PATH         | ${JAVA_HOME}/bin |

Testing the Java XDK Environment on UNIX

Table 3–3 describes the command-line utilities included in the Java XDK on UNIX. Before you can use these utilities, you must set up your environment.

Table 3–3  Java XDK Utilities

<table>
<thead>
<tr>
<th>Executable/Class</th>
<th>Directory/JAR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsql</td>
<td>$ORACLE_HOME/bin</td>
<td>XSQL command-line utility. The script executes the oracle.xml.xsql.XSQLCommandLine class. Edit this shell script for your environment before use. See Also: “Using the XSQL Pages Command-Line Utility” on page 14-11</td>
</tr>
<tr>
<td>OracleXML</td>
<td>$ORACLE_HOME/lib/xsu12.jar</td>
<td>XSU command-line utility See Also: “Using the XSU Command-Line Utility” on page 11-14</td>
</tr>
<tr>
<td>orajAXB</td>
<td>$ORACLE_HOME/bin</td>
<td>JAXB command-line utility See Also: “Using the JAXB Class Generator Command-Line Utility” on page 8-8</td>
</tr>
</tbody>
</table>
If your environment is set up correctly, then the UNIX shell script shown in Example 3–2 should generate version and usage information for the utilities.

**Example 3–2 Testing the Java XDK Environment on UNIX**

```bash
#!/usr/bin/tcsh
echo;echo "BEGIN TESTING";echo
echo;echo "now testing the XSQL utility...";echo
xsql
echo;echo "now testing the XSU utility...";echo
java OracleXML
echo;echo "now testing the JAXB utility...";echo
orajaxb -version
echo;echo "now testing the Pipeline utility...";echo
orapipe -version
echo;echo "now testing the XSLT Processor utility...";echo
oraxsl
echo;echo "now testing the TransX utility...";echo
transx
echo;echo "END TESTING"
```

**Setting Java XDK Environment Variables for Windows**

Table 3–4 describes the Windows environment variables required for use with the Java XDK components.
Testing the Java XDK Environment on Windows

Table 3–5 describes the command-line utilities included in the Java XDK on Windows. Before you can use these utilities, you must set up your environment.

Table 3–5  Java XDK Utilities

<table>
<thead>
<tr>
<th>Batch File/Class</th>
<th>Directory/JAR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsql.bat</td>
<td>%ORACLE_HOME%\bin</td>
<td>XSQL command-line utility. The batch file executes the oracle.xml.xsql.XSQLCommandLine class. Edit the batch file for your environment before use. [See Also: &quot;Using the XSQL Pages Command-Line Utility&quot; on page 14-11]</td>
</tr>
<tr>
<td>OracleXML</td>
<td>%ORACLE_HOME%\lib\xsu12.jar</td>
<td>XSU command-line utility [See Also: &quot;Using the XSU Command-Line Utility&quot; on page 11-14]</td>
</tr>
<tr>
<td>orajxb.bat</td>
<td>%ORACLE_HOME%\bin</td>
<td>JAXB command-line utility [See Also: &quot;Using the JAXB Class Generator Command-Line Utility&quot; on page 8-8]</td>
</tr>
<tr>
<td>orapipe.bat</td>
<td>%ORACLE_HOME%\bin</td>
<td>Pipeline command-line utility [See Also: &quot;Using the XML Pipeline Processor Command-Line Utility&quot; on page 9-8]</td>
</tr>
<tr>
<td>oraxml.bat</td>
<td>%ORACLE_HOME%\bin</td>
<td>XML parser command-line utility [See Also: &quot;Using the XML Parser Command-Line Utility&quot; on page 4-13]</td>
</tr>
<tr>
<td>oraxsl.bat</td>
<td>%ORACLE_HOME%\bin</td>
<td>XSLT processor command-line utility [See Also: &quot;Using the XSLT Processor Command-Line Utility&quot; on page 6-6]</td>
</tr>
<tr>
<td>transx.bat</td>
<td>%ORACLE_HOME%\bin</td>
<td>TransX command-line utility [See Also: &quot;Using the TransX Command-Line Utility&quot; on page 12-7]</td>
</tr>
</tbody>
</table>

If your environment is set up correctly, then you can run the commands in Example 3–3 at the system prompt to generate version and usage information for the utilities.

Table 3–4  Windows Environment Settings for Java XDK Components

<table>
<thead>
<tr>
<th>Variable</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%CLASSPATH%</td>
<td>Includes the following (note that a single period &quot;.&quot; to represent the current directory is not required but may be useful): .;%CLASSPATH%;%ORACLE_HOME%\lib\xmlparsev2.jar; %ORACLE_HOME%\lib\xsu12.jar;%ORACLE_HOME%\lib\xml.jar; %ORACLE_HOME%\lib\xmlmesg.jar;%ORACLE_HOME%\lib\oraclexsql.jar</td>
</tr>
<tr>
<td>%CLASSPATHJ%</td>
<td>For JDK 5, set as follows: CLASSPATHJ=%ORACLE_HOME%\jdbc\lib\ojdbc5.jar:%ORACLE_HOME%\lib\orai18n.jar</td>
</tr>
<tr>
<td>%JAVA_HOME%</td>
<td>Installation directory for the Java SDK, Standard Edition. Modify the path that links to the Java SDK.</td>
</tr>
<tr>
<td>%PATH%</td>
<td>%JAVA_HOME%\bin</td>
</tr>
</tbody>
</table>
Verifying the Java XDK Components Version

Example 3–3  Testing the Java XDK Environment on Windows

xsql.bat
java OracleXML
orajaxb.bat -version
orapipe.bat -version
oraxsl.bat
transx.bat

Verifying the Java XDK Components Version

To obtain the version of XDK you are working with, use javac to compile the Java code shown in Example 3–4.

Example 3–4  XDKVersion.java

```java
//
// XDKVersion.java
//
import java.net.URL;
import oracle.xml.parser.v2.XMLParser;
public class XDKVersion
{
    static public void main(String[] argv)
    {
        System.out.println("You are using version: ");
        System.out.println(XMLParser.getReleaseVersion());
    }
}
```

After compiling the source file with javac, run the program on the operating system command line as follows:

java XDKVersion
You are using version:
Oracle XML Developers Kit 11.1.0.6.0 - Production
This chapter contains these topics:

- Introduction to the XML Parsing for Java
- Using XML Parsing for Java: Overview
- Parsing XML with DOM
- Parsing XML with SAX
- Parsing XML with JAXP
- Compressing XML
- Tips and Techniques for Parsing XML

Introduction to the XML Parsing for Java

This section contains the following topics:

- Prerequisites
- Standards and Specifications
- XML Parsing in Java
- DOM in XML Parsing
- Scalable DOM
- SAX in the XML Parser
- JAXP in the XML Parser
- Namespace Support in the XML Parser
- Validation in the XML Parser
- Compression in the XML Parser

Prerequisites

Oracle XML parsing reads an XML document and uses DOM or SAX APIs to provide programmatic access to its content and structure. You can use parsing in validating or nonvalidating mode.

This chapter assumes that you are familiar with the following technologies:

- **Simple API for XML (SAX).** SAX is a standard for event-based XML parsing.
- **Java API for XML Processing (JAXP).** JAXP is a standard interface for processing XML with Java applications. It supports the DOM and SAX standards.
- **Document Type Definition (DTD).** An XML DTD defines the legal structure of an XML document.
- **XML Schema.** Like a DTD, an XML schema defines the legal structure of an XML document.
- **XML Namespaces.** Namespaces are a mechanism for differentiating element and attribute names.
- **Binary XML.** Both scalable and nonscalable DOMs can save XML documents in this format.

If you require a general introduction to the preceding technologies, consult the XML resources listed in "Related Documents" on page xxxvii of the preface.

### Standards and Specifications

The DOM Level 1, Level 2, and Level 3 specifications are W3C Recommendations. You can find links to the specifications for all three levels at the following URL:

http://www.w3.org/DOM/DOMTR

SAX is available in version 1.0, which is deprecated, and 2.0. It is not a W3C specification. You can find the documentation for SAX at the following URL:

http://www.saxproject.org/

XML Namespaces are a W3C Recommendation. You can find the specification at the following URL:

http://www.w3.org/TR/REC-xml-names

JCR 1.0 (also known as JSR 170) defines a standard Java API for applications to interact with content repositories.

**See Also:** *Oracle XML DB Developer’s Guide*,

JAXP version 1.2 includes an XSLT framework plus some updates to the parsing API to support DOM Level 2 and SAX version 2.0 and an improved scheme to locate pluggable implementations. JAXP provides support for XML schema and an XSLT compiler. You can access the JAXP specification, which is produced by Sun Microsystems, at the following URL:


**See Also:** Chapter 31, "XDK Standards" for an account of the standards supported by the XDK

### Large Node Handling

DOM Stream access to XML nodes is done by PL/SQL and Java APIs. Nodes in an XML document can now exceed 64 KBytes by a large amount. Thus JPEG, Word, PDF, RTF, and HTML documents can be more readily stored.

**See Also:** *Oracle XML DB Developer’s Guide* for complete details on the Java large node capabilities
XML Parsing in Java

XMLParser is the abstract base class for the XML parser for Java. An instantiated parser invokes the parse() method to read an XML document.

XMLDOMImplementation factory methods provide another method to parse Binary XML to create scalable DOM.

Figure 4–1 illustrates the basic parsing process, using XMLParser. The diagram does not apply to XMLDOMImplementation().

The following APIs provide a Java application with access to a parsed XML document:

- DOM API, which parses XML documents and builds a tree representation of the documents in memory. Use either a DOMParser object to parse with DOM or the XMLDOMImplementation interface factory methods to create a pluggable, scalable DOM.

- SAX API, which processes an XML document as a stream of events, which means that a program cannot access random locations in a document. Use a SAXParser object to parse with SAX.

- JAXP, which is a Java-specific API that supports DOM, SAX, and XSL. Use a DocumentBuilder or SAXParser object to parse with JAXP.

The sample XML document in Example 4–1 helps illustrate the differences among DOM, SAX, and JAXP.

Example 4–1 Sample XML Document

```xml
<?xml version="1.0"?>
<EMPLIST>
  <EMP>
    <ENAME>MARY</ENAME>
  </EMP>
  <EMP>
    <ENAME>SCOTT</ENAME>
  </EMP>
</EMPLIST>
```
DOM in XML Parsing

DOM builds an in-memory tree representation of the XML document. For example, the DOM API receives the document described in Example 4–1 and creates an in-memory tree as shown in Figure 4–2. DOM provides classes and methods to navigate and process the tree.

In general, the DOM API provides the following advantages:

- DOM API is easier to use than SAX because it provides a familiar tree structure of objects.
- Structural manipulations of the XML tree, such as re-ordering elements, adding to and deleting elements and attributes, and renaming elements, can be performed.
- Interactive applications can store the object model in memory, enabling users to access and manipulate it.
- DOM as a standard does not support XPath. However, most XPath implementations use DOM. The Oracle XDK includes DOM API extensions to support XPath.
- A pluggable, scalable DOM can be created that considerably improves scalability and efficiency.

DOM Creation

In Java XDK, there are three ways to create a DOM:

- Parse a document using DOMParser. This has been the traditional XDK approach.
- Create a scalable DOM using XMLDOMImplementation factory methods.
- Use an XMLDocument constructor. This is not a common solution in XDK.

Scalable DOM

With Oracle 11g Release 1 (11.1), XDK provides scalable, pluggable support for DOM. This relieves problems of memory inefficiency, limited scalability, and lack of control over the DOM configuration.

For the scalable DOM, the configuration and creation are mainly supported using the XMLDOMImplementation class.

These are important aspects of scalable DOM:

- Plug-in Data allows external XML representation to be directly used by Scalable DOM without replicating XML in internal representation.
- Scalable DOM is created on top of plug-in XML data through the Reader and InfosetWriter abstract interfaces. XML data can be in different forms, such as Binary XML, XMLType, and third-party DOM, and so on.
- Transient nodes. DOM nodes are created lazily and may be freed if not in use.
- Binary XML

The scalable DOM can use binary XML as both input and output format. Scalable DOM can interact with the data in two ways:

- Through the abstract InfosetReader and InfosetWriter interfaces. Users can (1) use the BinXML implementation of InfosetReader and InfosetWriter to
read and write BinXML data, and (2) use other implementations supplied by
the user to read and write in other forms of XML infoset.

- Through an implementation of the InfosetReader and InfosetWriter
  adaptor for BinXMLStream.

See Also: Chapter 5, "Using Binary XML for Java"

Scalable DOM support consists of the following:

- Pluggable DOM Support
- Lazy Materialization
- Configurable DOM Settings

Pluggable DOM Support
Pluggable DOM is an XDK mechanism that enables you to split the DOM API from the
data layer. The DOM API is separated from the data by the InfosetReader and
InfosetWriter interfaces.

Using pluggable DOM, XML data can be easily moved from one processor to another.
The DOM API includes unified standard APIs on top of the data to support node
access, navigation, update processes, and searching capability.

See Also: "Using Pluggable DOM" on page 4-19

Lazy Materialization
Using the lazy materialization mechanism, XDK only creates nodes that are accessed
and frees unused nodes from memory. Applications can process very large XML
documents with improved scalability.

See Also: "Using Lazy Materialization" on page 4-20

Configurable DOM Settings
DOM configurations can be made to suit different applications. You can configure the
DOM with different access patterns such as read-only, streaming, transient update,
and shadow copy, achieving maximum memory use and performance in your
applications.

See Also: "Using Configurable DOM Settings" on page 4-22

SAX in the XML Parser
Unlike DOM, SAX is event-based, so it does not build in-memory tree representations
of input documents. SAX processes the input document element by element and can
report events and significant data to callback methods in the application. The XML
document in Example 4–1 is parsed as a series of linear events as shown in Figure 4–2.

In general, the SAX API provides the following advantages:

- It is useful for search operations and other programs that do not need to
  manipulate an XML tree.
- It does not consume significant memory resources.
- It is faster than DOM when retrieving XML documents from a database.
**Figure 4–2  Comparing DOM (Tree-Based) and SAX (Event-Based) APIs**

<table>
<thead>
<tr>
<th>XML Document</th>
<th>The DOM interface creates a TREE structure based on the XML Document</th>
<th>The SAX interface creates a series of linear events based on the XML document</th>
</tr>
</thead>
</table>
| `<?xml version = "1.0"?>
<EMPLIST>
  <EMP>
    <ENAME>MARY</ENAME>
  </EMP>
<EMP>
  <ENAME>SCOTT</ENAME>
</EMPLIST>` | | `<?xml version="1.0"?>
<EMPLIST>
  <EMP>
    <ENAME>MARY</ENAME>
  </EMP>
  <EMP>
    <ENAME>SCOTT</ENAME>
</EMPLIST>` |

**JAXP in the XML Parser**

The JAXP API enables you to plug in an implementation of the SAX or DOM parser. The SAX and DOM APIs provided in the Oracle XDK are examples of vendor-specific implementations supported by JAXP.

In general, the advantage of JAXP is that you can use it to write interoperable applications. If an application uses features available through JAXP, then it can very easily switch the implementation.

The main disadvantage of JAXP is that it runs more slowly than vendor-specific APIs. In addition, several features are available through Oracle-specific APIs that are not available through JAXP APIs. Only some of the Oracle-specific features are available through the extension mechanism provided in JAXP. If an application uses these extensions, however, then the flexibility of switching implementation is lost.

**Namespace Support in the XML Parser**

The XML parser for Java can parse unqualified element types and attribute names as well as those in namespaces. Namespaces are a mechanism to resolve or avoid name collisions between element types or attributes in XML documents by providing "universal" names. Consider the XML document shown in **Example 4–2**.

**Example 4–2  Sample XML Document Without Namespaces**

```
<?xml version='1.0'?>
<addresslist>
  <company>
    <address>500 Oracle Parkway, Redwood Shores, CA 94065</address>
  </company>
<!-- ... -->
  <employee>
    <lastname>King</lastname>
    <address>3290 W Big Beaver</address>
  </employee>
</addresslist>
```
Without the use of namespaces, an application processing the XML document in Example 4–2 does not know whether the <address> tag refers to a company or employee address. As shown in Example 4–3, you can use namespaces to distinguish the <address> tags. The example declares the following XML namespaces:

http://www.oracle.com/employee
http://www.oracle.com/company

Example 4–3 associates the com prefix with the first namespace and the emp prefix with the second namespace. Thus, an application can distinguish <com:address> from <emp:address>.

**Example 4–3  Sample XML Document with Namespaces**

```xml
<?xml version='1.0'?>
<addresslist>
<!-- ... -->
<com:company
xmlns:com="http://www.oracle.com/company">
  <com:address>500 Oracle Parkway,
  Redwood Shores, CA 94065</com:address>
</com:company>
<!-- ... -->
<emp:employee
xmlns:emp="http://www.oracle.com/employee">
  <emp:lastname>King</emp:lastname>
  <emp:address>3290 W Big Beaver
  Troy, MI 48084</emp:address>
</emp:employee>
</addresslist>
```

It is helpful to remember the following terms when parsing documents that use namespaces:

- **Namespace prefix**, which is a namespace prefix declared with xmlns. In Example 4–3, emp and com are namespace prefixes.
- **Local name**, which is the name of an element or attribute without the namespace prefix. In Example 4–3, employee and company are local names.
- **Qualified name**, which is the local name plus the prefix. In Example 4–3, emp:employee and com:company are qualified names.
- **Namespace URI**, which is the URI assigned to xmlns. In Example 4–3, http://www.oracle.com/employee and http://www.oracle.com/company are namespace URIs.
Validation in the XML Parser

Applications invoke the `parse()` method to parse XML documents. Typically, applications invoke initialization and termination methods in association with the `parse()` method. You can use the `setValidationMode()` method defined in `oracle.xml.parser.v2.XMLParser` to set the parser mode to validating or nonvalidating.

By parsing an XML document according to the rules specified in a DTD or an XML schema, a validating XML parser determines whether the document conforms to the specified DTD or XML schema. If the XML document does conform, then the document is valid, which means that the structure of the document conforms to the DTD or schema rules. A nonvalidating parser checks for well-formedness only.

Configuring the Validation Mode

Table 4–1 shows the flags that you can use in `setValidationMode()` to set the validation mode in the Oracle XDK parser.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>The XML Parser ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonvalidating</td>
<td>NONVALIDATING</td>
<td>Verifies that the XML is well-formed and parses the data.</td>
</tr>
<tr>
<td>DTD validating</td>
<td>DTD_VALIDATION</td>
<td>Verifies that the XML is well-formed and validates the XML data against the DTD. The DTD defined in the <code>&lt;!DOCTYPE&gt;</code> declaration must be relative to the location of the input XML document.</td>
</tr>
<tr>
<td>Schema validation</td>
<td>SCHEMA_VALIDATION</td>
<td>Validates the XML Document according to the XML schema specified for the document.</td>
</tr>
<tr>
<td>LAX validation</td>
<td>SCHEMA_LAX_VALIDATION</td>
<td>Tries to validate part or all of the instance document as long as it can find the schema definition. It does not raise an error if it cannot find the definition. See the sample program <code>XSDLax.java</code> in the schema directory.</td>
</tr>
<tr>
<td>Strict validation</td>
<td>SCHEMA_STRICT_VALIDATION</td>
<td>Tries to validate the whole instance document, raising errors if it cannot find the schema definition or if the instance does not conform to the definition.</td>
</tr>
<tr>
<td>Partial validation</td>
<td>PARTIAL_VALIDATION</td>
<td>Validates all or part of the input XML document according to the DTD, if present. If the DTD is not present, then the parser is set to nonvalidating mode.</td>
</tr>
<tr>
<td>Auto validation</td>
<td>AUTO_VALIDATION</td>
<td>Validates all or part of the input XML document according to the DTD or XML schema, if present. If neither is present, then the parser is set to nonvalidating mode.</td>
</tr>
</tbody>
</table>

In addition to setting the validation mode with `setValidationMode()`, you can use the `oracle.xml.parser.schema.XSDBuilder` class to build an XML schema and then configure the parser to use it by invoking the `XMLParser.setXMLSchema()` method. In this case, the XML parser automatically sets the validation mode to `SCHEMA_STRICT_VALIDATION` and ignores the `schemaLocation` and `noNamespaceSchemaLocation` attributes. You can also change the validation mode to `SCHEMA_LAX_VALIDATION`. The `XMLParser.setDoctype()` method is a parallel method for DTDs, but unlike `setXMLSchema()` it does not alter the validation mode.

See Also:

- Chapter 7, "Using the Schema Processor for Java" on page 7-1 to learn about validation
- Oracle Database XML Java API Reference to learn about the `XMLParser` and `XSDBuilder` classes
Compression in the XML Parser

You can use the XML compressor, which is implemented in the XML parser, to compress and decompress XML documents. The compression algorithm is based on tokenizing the XML tags. The assumption is that any XML document repeats a number of tags and so tokenizing these tags gives considerable compression. The degree of compression depends on the type of document: the larger the tags and the lesser the text content, the better the compression.

The Oracle XML parser generates a binary compressed output from an in-memory DOM tree or SAX events generated from an XML document. Table 4–2 describes the two types of compression.

### Table 4–2 XML Compression with DOM and SAX

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Compression APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM-based</td>
<td>The goal is to reduce the size of the XML document without losing the structural and hierarchical information of the DOM tree. The parser serializes an in-memory DOM tree, corresponding to a parsed XML document, and generates a compressed XML output stream. The serialized stream regenerates the DOM tree when read back.</td>
<td>Use the <code>writeExternal()</code> method to generate compressed XML and the <code>readExternal()</code> method to reconstruct it. The methods are in the <code>oracle.xml.parser.v2.XMLDocument</code> class.</td>
</tr>
<tr>
<td>SAX-based</td>
<td>The SAX parser generates a compressed stream when it parses an XML file. SAX events generated by the SAX parser are handled by the SAX compression utility, which generates a compressed binary stream. When the binary stream is read back, the SAX events are generated.</td>
<td>To generate compressed XML, instantiate <code>oracle.xml.comp.CXMLHandlerBase</code> by passing an output stream to the constructor. Pass the object to <code>SAXParser.setContentHandler()</code> and then execute the <code>parse()</code> method. Use the <code>oracle.xml.comp.CXMLParser</code> class to decompress the XML. Note: <code>CXMLHandlerBase</code> implements both SAX 1.0 and 2.0, but because 1.0 is deprecated, it is recommended that you use the 2.0 API.</td>
</tr>
</tbody>
</table>

The compressed streams generated from DOM and SAX are compatible, that is, you can use the compressed stream generated from SAX to generate the DOM tree and vice versa. As with XML documents in general, you can store the compressed XML data output in the database as a `BLOB`.

When a program parses a large XML document and creates a DOM tree in memory, it can affect performance. You can compress an XML document into a binary stream by serializing the DOM tree. You can regenerate the DOM tree without validating the XML data in the compressed stream. You can treat the compressed stream as a serialized stream, but the data in the stream is more controlled and managed than the compression implemented by Java's default serialization.

**Note:** Oracle Text cannot search a compressed XML document. Decompression reduces performance. If you are transferring files between client and server, then HTTP compression can be easier.

Using XML Parsing for Java: Overview

The fundamental component of any XML development is XML parsing. XML parsing for Java is a standalone XML component that parses an XML document (and possibly also a standalone DTD or XML Schema) so that your program can process it. This section contains the following topics:
Using the XML Parser for Java: Basic Process

Figure 4–3 shows how to use the XML parser in a typical XML processing application.

The basic process of the application shown in Figure 4–3 is as follows:

1. The DOM or SAX parser parses input XML documents. For example, the program can parse XML data documents, DTDs, XML schemas, and XSL stylesheets.

2. If you implement a validating parser, then the processor attempts to validate the XML data document against any supplied DTDs or XML schemas.

See Also:

- Chapter 6, "Using the XSLT Processor for Java"
- Chapter 7, "Using the Schema Processor for Java" on page 7-1 to learn about validation
- Oracle Database XML Java API Reference for XML parser classes and methods

Running the XML Parser Demo Programs

Demo programs for the XML parser for Java are included in $ORACLE_HOME/xdk/demo/java/parser. The demo programs are distributed among the subdirectories described in Table 4–3.
<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
<th>These programs ...</th>
</tr>
</thead>
</table>
| **common** | class.xml
DemoUtil.java
empl.xml
family.dtd
family.xml
iden.xsl
NSExample.xml
traversal.xml | Provide XML files and Java programs for general use with the XML parser. For example, you can use the XSLT stylesheet iden.xsl to achieve an identity transformation of the XML files. DemoUtil.java implements a helper method to create a URL from a file name. This method is used by many of the other demo programs. |
| **comp** | DOMCompression.java
DOMDeCompression.java
SAXCompression.java
SAXDeCompression.java
SampleSAXHandler.java
sample.xml
xml.ser | Illustrate DOM and SAX compression: |
| | | ■ DOMCompression.java compresses a DOM tree. |
| | | ■ DOMDeCompression.java reads back a DOM from a compressed stream. |
| | | ■ SAXCompression.java compresses the output from a SAX parser. |
| | | ■ SAXDeCompression.java regenerates SAX events from the compressed stream. |
| | | ■ SampleSAXHandler.java illustrates use of a handler to handle the events thrown by the SAX DeCompressor. |
| **dom** | AutoDetectEncoding.java
DOM2Namespace.java
DOMNamespace.java
DOMRangeSample.java
DOMSample.java
EventSample.java
I18nSafeXMLFileWritingSample.java
NodeIteratorSample.java
ParseXMLFromString.java
TreeWalkerSample.java | Illustrate uses of the DOM API: |
| | | ■ DOM2Namespace.java shows how to use DOM Level 2.0 APIs. |
| | | ■ DOMNamespace.java shows how to use Namespace extensions to DOM APIs. |
| | | ■ DOMRangeSample.java shows how to use DOM Range APIs. |
| | | ■ DOMSample.java shows basic use of the DOM APIs. |
| | | ■ EventSample.java shows how to use DOM Event APIs. |
| | | ■ NodeIteratorSample.java shows how to use DOM Iterator APIs. |
| | | ■ TreeWalkerSample.java shows how to use DOM TreeWalker APIs. |
Documentation for how to compile and run the sample programs is located in the README. The basic steps are as follows:

1. Change into the $ORACLE_HOME/xdk/demo/java/parser directory (UNIX) or %ORACLE_HOME%/xdk/demo/java/parser directory (Windows).

2. Set up your environment as described in "Setting Up the Java XDK Environment" on page 3-5.

3. Change into each of the following subdirectories and run make (UNIX) or Make.bat (Windows) at the command line. For example:

   cd comp;make;cd ..
   cd jaxp;make;cd ..
   cd sax;make;cd ..
   cd dom;make;cd ..
   cd xslt;make;cd ..
The make file compiles the source code in each directory, runs the programs, and writes the output for each program to a file with an *.out extension.

4. You can view the *.out files to view the output for the programs.

Using the XML Parser Command-Line Utility

The oraxml utility, which is located in $ORACLE_HOME/bin (UNIX) or $ORACLE_HOME\bin (Windows), is a command-line interface that parses XML documents. It checks for both well-formedness and validity.

To use oraxml ensure that the following is true:

- Your CLASSPATH is set up as described in "Setting Up the Java XDK Environment" on page 3-5. In particular, make sure you CLASSPATH environment variable points to the xmlparserv2.jar file.
- Your PATH environment variable can find the Java interpreter that comes with the version of the JDK that you are using.

Table 4–4 lists the oraxml command-line options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-help</td>
<td>Prints the help message</td>
</tr>
<tr>
<td>-version</td>
<td>Prints the release version</td>
</tr>
<tr>
<td>-novalidate fileName</td>
<td>Checks whether the input file is well-formed</td>
</tr>
<tr>
<td>-dtd fileName</td>
<td>Validates the input file with DTD Validation</td>
</tr>
<tr>
<td>-schema fileName</td>
<td>Validates the input file with Schema Validation</td>
</tr>
<tr>
<td>-log logfile</td>
<td>Writes the errors to the output log file</td>
</tr>
<tr>
<td>-comp fileName</td>
<td>Compresses the input XML file</td>
</tr>
<tr>
<td>-decomp fileName</td>
<td>Decompresses the input compressed file</td>
</tr>
<tr>
<td>-enc fileName</td>
<td>Prints the encoding of the input file</td>
</tr>
<tr>
<td>-warning</td>
<td>Show warnings</td>
</tr>
</tbody>
</table>

For example, change into the $ORACLE_HOME/xdk/demo/java/parser/common directory. You can validate the document family.xml against family.dtd by executing the following on the command line:

oraxml -dtd -enc family.xml

The output should appear as follows:

The encoding of the input file: UTF-8

The input XML file is parsed without errors using DTD validation mode.

Parsing XML with DOM

The W3C standard library org.w3c.dom defines the Document class as well as classes for the components of a DOM. The Oracle XML parser includes the standard DOM APIs and is compliant with the W3C DOM recommendation. Along with org.w3c.dom, Oracle XML parsing includes classes that implement the DOM APIs and extend them...
to provide features such as printing document fragments and retrieving namespace information.

This section contains the following topics:

- Using the DOM API
- DOM Parser Architecture
- Performing Basic DOM Parsing
- Creating Scalable DOM
- Performing DOM Operations with Namespaces
- Performing DOM Operations with Events
- Performing DOM Operations with Ranges
- Performing DOM Operations with TreeWalker

Using the DOM API

To implement DOM-based components in your XML application, you can use the following XDK classes:

- `oracle.xml.parser.v2.DOMParser`. This class implements an XML 1.0 parser according to the W3C recommendation. Because DOMParser extends XMLParser, all methods of XMLParser are available to DOMParser.

  **See Also:** "DOM Parser Architecture" on page 4-14

- `oracle.xml.parser.v2.XMLDOMImplementation`. This class contains factory methods used to created scalable, pluggable DOM.

  For purposes of this discussion, DOMs created with the XMLDOMImplementation class are referred to as scalable or pluggable DOM.

  **See Also:** "Creating Scalable DOM" on page 4-19

You can also make use of the DOMNamespace and DOM2Namespace classes, which are sample programs included in $ORACLE_HOME/xdk/demo/java/parser/dom.

DOM Parser Architecture

Figure 4–4 is an architectural diagram of the DOM Parser.
Performing Basic DOM Parsing

The program DOMSample.java is provided to illustrate the basic steps for parsing an input XML document and accessing it through a DOM.

The program receives an XML file as input, parses it, and prints the elements and attributes in the DOM tree.

The steps provide reference to tables that provide possible methods and interfaces you can use at that point.

1. Create a DOMParser object by calling the DOMParser() constructor. You can use this parser to parse input XML data documents as well as DTDs. The following code fragment from DOMSample.java illustrates this technique:

   ```java
   DOMParser parser = new DOMParser();
   ```

2. Configure parser properties. See Table 4–5 on page 4-18.

   The following code fragment from DOMSample.java specifies the error output stream, sets the validation mode to DTD validation, and enables warning messages:

   ```java
   parser.setErrorStream(System.err);
   parser.setValidationMode(DOMParser.DTD_VALIDATION);
   parser.showWarnings(true);
   ```
3. Parse the input XML document by invoking the `parse()` method. The program builds a tree of `Node` objects in memory.

This code fragment from `DOMSample.java` shows how to parse an instance of the `java.net.URL` class:

```java
parser.parse(url);
```

Note that the XML input can be a file, string buffer, or URL. As illustrated by the following code fragment, `DOMSample.java` accepts a filename as a parameter and calls the `createURL` helper method to construct a URL object that can be passed to the parser:

```java
public class DOMSample
{
    static public void main(String[] argv)
    {
        try
        {
            if (argv.length != 1)
            {
                // Must pass in the name of the XML file.
                System.err.println("Usage: java DOMSample filename");
                System.exit(1);
            }
            ...
            // Generate a URL from the filename.
            URL url = DemoUtil.createURL(argv[0]);
            ...
        }
    }
}
```

4. Invoke `getDocument()` to obtain a handle to the root of the in-memory DOM tree, which is an `XMLDocument` object. You can use this handle to access every part of the parsed XML document. The `XMLDocument` class implements the interfaces shown in Table 4–6.

This code fragment from `DOMSample.java` illustrates this technique:

```java
XMLDocument doc = parser.getDocument();
```

5. Obtain and manipulate DOM nodes of the retrieved document by calling various `XMLDocument` methods. See Table 4–7.

The following code fragment from `DOMSample.java` uses the `DOMParser.print()` method to print the elements and attributes of the DOM tree:

```java
System.out.print("The elements are: ");
printElements(doc);
System.out.println("The attributes of each element are: ");
printElementAttributes(doc);
```

The program implements the `printElements()` method, which calls `getElementsByTagName()` to obtain a list of all the elements in the DOM tree. It then loops through each item in the list and calls `getNodeName()` to print the name of each element:

```java
static void printElements(Document doc)
{
    NodeList nl = doc.getElementsByTagName("*");
    Node n;
    for (int i=0; i<nl.getLength(); i++)
    {
```
The program implements the `printElementAttributes()` method, which calls `Document.getElementsByTagName()` to obtain a list of all the elements in the DOM tree. It then loops through each element in the list and calls `Element.getAttributes()` to obtain the list of attributes for the element. It then calls `Node.getNodeName()` to obtain the attribute name and `Node.getNodeValue()` to obtain the attribute value:

```java
static void printElementAttributes(Document doc)
{
    NodeList nl = doc.getElementsByTagName($$);
    Element e;
    Node n;
    NamedNodeMap nnm;

    String attrname;
    String attrval;
    int i, len;

    len = nl.getLength();
    for (int j=0; j < len; j++)
    {
        e = (Element)nl.item(j);
        System.out.println(e.getTagName() + ":");
        nnm = e.getAttributes();
        if (nnm != null)
        {
            for (i=0; i<nnm.getLength(); i++)
            {
                n = nnm.item(i);
                attrname = n.getNodeName();
                attrval = n.getNodeValue();
                System.out.print(" " + attrname + " = " + attrval);
            }
        }
        System.out.println();
    }
}
```

6. Reset the parser state by invoking the `reset()` method. The parser is now ready to parse a new document.

### Useful Methods and Interfaces

The following tables provide useful methods and interfaces to use in creating an application such as the one just created in "Performing Basic DOM Parsing" on page 4-15.

Table 4–5 lists useful configuration methods.
Table 4–5  DOMParser Configuration Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Use this method to . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBaseURL()</td>
<td>Set the base URL for loading external entities and DTDs. Call this method if the XML document is an InputStream.</td>
</tr>
<tr>
<td>setDoctype()</td>
<td>Specify the DTD to use when parsing.</td>
</tr>
<tr>
<td>setErrorStream()</td>
<td>Create an output stream for the output of errors and warnings.</td>
</tr>
<tr>
<td>setPreserveWhitespace()</td>
<td>Instruct the parser to preserve the whitespace in the input XML document.</td>
</tr>
<tr>
<td>setValidationMode()</td>
<td>Set the validation mode of the parser. Table 4–1 describes the flags that you can use with this method.</td>
</tr>
<tr>
<td>showWarnings()</td>
<td>Specify whether the parser should print warnings.</td>
</tr>
</tbody>
</table>

Table 4–6 lists the interfaces that the XMLDocument class implements.

Table 4–6  Some Interfaces Implemented by XMLDocument

<table>
<thead>
<tr>
<th>Interface</th>
<th>Defines . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.w3c.dom.Node</td>
<td>A single node in the document tree and methods to access and process the node.</td>
</tr>
<tr>
<td>org.w3c.dom.Document</td>
<td>A Node that represents the entire XML document.</td>
</tr>
<tr>
<td>org.w3c.dom.Element</td>
<td>A Node that represents an XML element.</td>
</tr>
</tbody>
</table>

Table 4–7 lists some useful methods for obtaining nodes.

Table 4–7  Useful XMLDocument Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Use this method to . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAttributes()</td>
<td>Generate a NamedNodeMap containing the attributes of this node (if it is an element) or null otherwise.</td>
</tr>
<tr>
<td>getElementsByTagName()</td>
<td>Retrieve recursively all elements that match a given tag name under a certain level. This method supports the * tag, which matches any tag. Call getElementsByTagName(&quot;*&quot;) through the handle to the root of the document to generate a list of all elements in the document.</td>
</tr>
<tr>
<td>getExpandedName()</td>
<td>Obtain the expanded name of the element. This method is specified in the NSName interface.</td>
</tr>
<tr>
<td>getLocalName()</td>
<td>Obtain the local name for this element. If an element name is &lt;E1:locn xmlns:E1=&quot;<a href="http://www.oracle.com/%3E">http://www.oracle.com/&gt;</a>&quot;, then locn is the local name.</td>
</tr>
<tr>
<td>getNamespaceURI()</td>
<td>Obtain the namespace URI of this node, or null if it is unspecified. If an element name is &lt;E1:locn xmlns:E1=&quot;<a href="http://www.oracle.com/%3E">http://www.oracle.com/&gt;</a>&quot;, then <a href="http://www.oracle.com">http://www.oracle.com</a> is the namespace URI.</td>
</tr>
<tr>
<td>getNodeName()</td>
<td>Obtain the name of a node in the DOM tree.</td>
</tr>
<tr>
<td>getNodeValue()</td>
<td>Obtain the value of this node, depending on its type. This mode is in the Node interface.</td>
</tr>
<tr>
<td>getPrefix()</td>
<td>Obtain the namespace prefix for an element.</td>
</tr>
</tbody>
</table>
Creating Scalable DOM

This section discusses how to create and use a scalable DOM.

This section contains the following topics:

- Using Pluggable DOM
- Using Lazy Materialization
- Using Configurable DOM Settings
- Scalable DOM Applications

Using Pluggable DOM

Pluggable DOM has the DOM API split from the data. The underlying data can be either internal or plug-in, and both can be in binary XML.

- Internal Data
  
  To plug in internal data (XML text that has not been parsed), the XML text must be saved as binary XML, then parsed by the DOMParser. The parsed binary XML can be then be plugged into the InfoSetReader of the DOM API layer.
  
  The InfoSetReader argument is the interface to the underlying XML data.

- Plug-in Data
  
  Plug-in data is data that has already been parsed and therefore can be transferred from one processor to another without requiring parsing.

To create a pluggable DOM, XML data is plugged in through the InfoSetReader interface on an XMLDOMImplementation object, for example:

```java
public Document createDocument(InfosetReader reader) throws DOMException
```

The InfoSetReader API is implemented on top of the XDK BinXMLStream. Optional adaptors for other forms of XML data such as DOM4J, JDOM, or JDBC may also be supported. Users can also plug in their own implementations.

InfoSetReader serves as the interface between the scalable DOM API layer and the underlying data. It is a generic, stream-based pull API that accesses XML data. The InfoSetReader retrieves sequential events from the XML stream and queries the state and data from these events. In the following example, the XML data is scanned to retrieve the QName and attributes of all elements:

```java
InfosetReader reader;
While {reader.hasNext()}
{
    reader.next();
    if {reader.getEventType() == START_ELEMENT}
    {
        QName name = reader.getQName();
        TypedAttributeList attrList = reader.getAttributeList();
    }
```

Table 4–7 (Cont.) Useful XMLDocument Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Use this method to . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>getQualifiedName()</td>
<td>Obtain the qualified name for an element. If an element name is &lt;E1:locn xmlns:E1=&quot;<a href="http://www.oracle.com/*%22%3E">http://www.oracle.com/*&quot;&gt;</a>, then E1:locn is the qualified name.</td>
</tr>
<tr>
<td>getTagName()</td>
<td>Obtain the name of an element in the DOM tree.</td>
</tr>
</tbody>
</table>

XML Parsing for Java 4-19
InfosetReader Options

The InfosetReader interface supports the following functionality:

Copying: To support shadow copy of DOM across documents, a new copy of InfosetReader can be created to ensure thread safety, using the Clone method. An InfosetReader obtained from BinXMLStream always supports this (Optional).

```
if (reader.hasSeekSupport())
    reader.seek(offset);
```

See Also: "Using Shadow Copy" on page 4-22

Moving Focus: To support lazy materialization, the InfosetReader may have the ability to move focus to any location specified by Offset (Optional).

```
InfosetWriter
```

See Also: "Using Lazy Materialization" on page 4-20

InfosetWriter

InfosetWriter is an extension of the InfosetReader interface that supports data writing. XDK provides an implementation on top of binary XML. Users cannot modify this implementation.

Saving XML Text as Binary XML

To create a scalable DOM from XML text, you must save the XML text as binary XML, before you can run DOMParser on it. You can save the XML text as either of the following:

- Binary XML
- References to binary XML: You can save the section reference of binary XML instead of actual data, if you know that the data source is available for deserialization.

The following example illustrates how to save as binary XML.

```
XMLDocument doc;
InfosetWriter writer;
doc.save(writer, false);
writer.close();
```

To save as references to binary XML, use true as the argument for the save command.

See Also: Chapter 5, "Using Binary XML for Java"

Using Lazy Materialization

Using lazy materialization, you can plug in an empty DOM, which can pull in more data when needed and free nodes when they are no longer needed.

Pulling Data on Demand

The plug-in DOM architecture creates an empty DOM, which contains a single Document node as the root of the tree. The rest of the DOM tree can be expanded later if it is accessed. A node may have unexpanded child and sibling nodes, but its parent and ancestors are always expanded. Each node maintains the InfosetReader.Offset property of the next node so that the DOM can pull data additional to create the next node.

Depending on the access method type, DOM nodes may expand more than the set of nodes returned:
- **DOM Navigation**
  
The DOM navigation interface allows access to neighboring nodes such as first child, last child, parent, previous or next sibling. If node creation is needed, it is always done in document order.

- **ID Indexing**
  
  A DTD or XML schema can specify nodes with the type ID. If the DOM supports ID indexing, those nodes can be directly retrieved using the index. In the case of scalable DOM, retrieval by index does not cause the expansion of all previous nodes, but their ancestor nodes are materialized.

- **XPath Expressions**
  
  XPath evaluation can cause materialization of all intermediate nodes in memory. For example, the descendent axis '//' results in the expansion of the whole subtree, although some nodes might be released after evaluation.

**Freeing Nodes When No Longer Needed**

Scalable DOM supports either manual or automatic dereferencing of nodes:

- **Automatic Dereferencing Using Weak References**

  To enable automatic dereferencing, set `PARTIAL_DOM` attribute to `Boolean.TRUE`.

  Supporting DOM navigation requires adding cross references among nodes. In automatic dereferencing mode, some of the links are weak references, which can be freed during garbage collection.

  Node release is based on the importance of the links: Links to parent nodes cannot be dropped because ancestors provide context for in-scope namespaces and it is difficult to retrieve dropped parent nodes using streaming APIs such as `InfosetReader`.

  The scalable DOM always holds its parent and previous sibling strongly but holds its children and following sibling weakly. When the Java Virtual Machine frees the nodes, references to them are still available in the underlying data so they can be recreated if needed.

- **Manual Dereferencing**

  To enable manual dereferencing, set the attribute `PARTIAL_DOM` to `Boolean.FALSE` and create the DOM with plug-in XML data.

  In this mode, the DOM depends on the application to explicitly dereference a document fragment from the whole tree. There are no weak references. It is recommended that if an application has a deterministic order of processing the data, to avoid the extra overhead of repeatedly releasing and recreating nodes.

  To dereference a node from all other nodes, call `freeNode()` on it. For example:

  ```java
  Element root = doc.getDocumentElement();
  Node item = root.getFirstChild();
  While (item != null)
  {
    processItem(item);
    Node tmp = item;
    item = item.getNextSibling();
    ((XMLNode)tmp).freeNode();
  }
  ```

  The `freeNode` call has no effect on a non-scalable DOM.
Note that dereferencing nodes is different from removing nodes from a DOM tree. The DOM tree does not change when `freeNode` is called on a DOM node. The node can still be accessed and recreated from its parent, previous, and following siblings. However, a variable that holds the node will throw an error when accessing the node after the node has been freed.

**Using Shadow Copy** With shadow copy, the data underneath can be shared to avoid data replications

Cloning, a common operation in XML processing, can be done lazily with pluggable DOM.

When the `copy` method is used, it creates just the root node of the fragment being copied, and the subtree can be expanded on demand.

Data sharing is for the underlying data, not the DOM nodes themselves. The DOM specification requires that the clone and its original have different node identities, and that they have different parent nodes.

**Incorporating DOM Updates**

The DOM API supports update operations such as adding, deleting nodes, setting, deleting, changing, and inserting values. When a DOM is created by plugging in XML data, the underlying data is considered external to the DOM. DOM updates are visible from the DOM APIs but the data source remains the same. Normal update operations are available and do not interfere with each other.

To make a modified DOM persistent, you must explicitly save the DOM. This merges all the changes with the original data and serializes the data in persistent storage. If you do not save a modified DOM explicitly, the changes are lost once the transaction ends.

**Using the PageManager Interface to Support Internal Data**

When XML text is parsed with `DOMParser` and configured to create a scalable DOM, internal data is cached in the form of binary XML, and the DOM API layer is built on top of the internal data. This provides increased scalability, because the binary XML is more compact than DOM nodes.

For additional scalability, the scalable DOM can use backend storage for binary data through the `PageManager` interface. Then, binary data can be swapped out of memory when not in use.

This code example illustrates how to use the `PageManager` interface.

```java
DOMParser parser = new DOMParser();
parser.setAttribute(PARTIAL_DOM, Boolean.TRUE); // enable scalable DOM
parser.setAttribute(PAGE_MANAGER, new FilePageManager("pageFile"));
...
// DOMParser other configuration
parser.parse(fileURL);
XMLDocument doc = parser.getDocument();
```

If the `PageManager` interface is not used, then the parser caches the whole document as binary XML.

**Using Configurable DOM Settings**

When you create a DOM with the `XMLDOMImplementation` class, you can configure the DOM to suit different applications and achieve maximum efficiency, using the `setAttribute` method in the `XMLDOMImplementation` class.
public void setAttribute(String name, Object value) throws 
IllegalArgumentException

For scalable DOM, call `setAttribute` for the `PARTIAL_DOM` and `ACCESS_MODE` attributes.

---

**Note:** New attribute values always affect the next DOM, not the current one, so an instance of `XMLDOMImplementation` can be used to create DOMs with different configurations.

---

- **PARTIAL_DOM**

  This attribute indicates whether the DOM is scalable (partial), and whether it takes a Boolean value. DOM creation is scalable when the attribute is set to `TRUE` and nodes that are not in use are freed and recreated when needed. DOM creation is not scalable when the attribute is set to `FALSE` or is not set.

- **ACCESS_MODE**

  This attribute controls the access of the DOM and applies to both scalable DOM and non-scalable DOM. It has the following values:

  - **UPDATEABLE**
    
    The DOM supports all DOM update operations.
    
    `UPDATEABLE` is the default value for the `ACCESS_MODE` attribute, in order to maintain backward compatibility with the XDK DOM implementation.

  - **READ_ONLY**
    
    DOM can only read this.
    
    Any attempt to modify the DOM tree results in an error, but node creation such as cloning is allowed, as long as the new nodes are not added to the DOM tree.

  - **FORWARD_READ**
    
    This value allows forward navigation, such as `getFirstChild()`, `getNextSibling()`, and `getLastChild()`, but not backward access, such as `getPreviousSibling()`.
    
    `FORWARD_READ` can still access parent and ancestor nodes.

  - **STREAMING**
    
    DOM access is limited to the stream of nodes in Document Order, similar to SAX-event access.
    
    Following the concept of current node in stream mode, the current node is the last node that has been accessed in document order. Applications can hold nodes in variables and revisit them, but using the DOM method to access any node before the current node causes a DOM error. However, accessing ancestor nodes and attribute nodes is always allowed.
    
    The following illustrates the DOM behavior in stream mode:

    ```java
    Node parent = currentNode.getParentNode(); // OK although parent is before current node
    Node child = parent.getFirstChild(); // Error if the current node is not the first child of parent!
    ```
Attribute attr = parent.getFirstAttribute(); // OK accessing attributes from
Element is always // allowed

The following lists the access modes from less restrictive to more restrictive.
UPDATEABLE > READ_ONLY > FORWARD_READ > STREAM_READ

Performance Advantages to Configurable DOM Settings
DOM cannot be modified in READ_ONLY mode, so the whole write buffer is not needed.
DOM does not read backward in FORWARD_READ mode, except to the ancestor node.
Therefore, the previous sibling link is not created.
DOM only maintains parent links and does not need to remember data location for a
node in STREAM_READ mode. Therefore, it does not need to recreate any node that has
been freed.

Scalable DOM Applications
Here is an application that creates and uses a scalable, pluggable DOM:

XMLDOMImplementation domimpl = new XMLDOMImplementation();
domimpl.setAttribute(XMLDocument.SCALABLE_DOM, Boolean.TRUE);
domimpl.setAttribute(XMLDocument.ACCESS_MODE, XMLDocument.UPDATEABLE);
XMLDocument scalableDoc = (XMLDocument) domimpl.createDocument(reader);

Here is an application that creates and uses a scalable, pluggable DOM based on
binary XML, which is described in Chapter 5, "Using Binary XML for Java":

BinXMLProcessor proc = BinXMLProcessorFactory.createProcessor();
BinXMLStream bstr = proc.createBinXMLStream();
BinXMLEncoder enc = bstr.getEncoder();
enc.setProperty(BinXMLEncoder.ENC_SCHEMA_AWARE, false);

SAXParser parser = new SAXParser();
parser.setContentHandler(enc.getContentHandler());
parser.setErrorHandler(enc.getErrorHandler());
parser.parse(BinXMLUtil.createURL(xmlfile));

BinXMLDecoder dec = bstr.getDecoder();
InfosetReader reader = dec.getReader();
XMLDOMImplementation domimpl = new XMLDOMImplementation();
XMLDocument currentDoc = (XMLDocument) domimpl.createDocument(reader);

Performing DOM Operations with Namespaces
The DOM2Namespace.java program illustrates a simple use of the parser and
namespace extensions to the DOM APIs. The program receives an XML document,
parses it, and prints the elements and attributes in the document.
The initial four steps of the "Performing Basic DOM Parsing" on page 4-15, from parser
creation to the getDocument() call, are basically the same as for DOM2Namespace.java.
The principal difference is in printing the DOM tree, which is step 5 on page 4-16. The
DOM2Namespace.java program does the following instead:

// Print document elements
printElements(doc);

// Print document element attributes
System.out.println("The attributes of each element are: ");
printElementAttributes(doc);

The printElements() method implemented by DOM2Namespace.java calls
getElementsByTagName() to obtain a list of all the elements in the DOM tree. It then
loops through each item in the list and casts each Element to an nsElement. For each
nsElement it calls nsElement.getPrefix() to get the namespace prefix,
nsElement.getLocalName() to get the local name, and nsElement.getNamespaceURI() to get the namespace URI:

static void printElements(Document doc)
{
    NodeList nl = doc.getElementsByTagName("*");
    Element nsElement;
    String prefix;
    String localName;
    String nsName;

    System.out.println("The elements are: ");
    for (int i=0; i < nl.getLength(); i++)
    {
        nsElement = (Element)nl.item(i);
        prefix = nsElement.getPrefix();
        System.out.println(" ELEMENT Prefix Name :" + prefix);
        localName = nsElement.getLocalName();
        System.out.println(" ELEMENT Local Name :" + localName);
        nsName = nsElement.getNamespaceURI();
        System.out.println(" ELEMENT Namespace :" + nsName);
    }
    System.out.println();
}

The printElementAttributes() method calls Document.getElementsByTagName() to
obtain a NodeList of the elements in the DOM tree. It then loops through each element
and calls Element.getAttributes() to obtain the list of attributes for the element as
special list called a NamedNodeMap. For each item in the attribute list it calls
nsAttr.getPrefix() to get the namespace prefix, nsAttr.getLocalName() to get the
local name, and nsAttr.getValue() to obtain the value:

static void printElementAttributes(Document doc)
{
    NodeList nl = doc.getElementsByTagName("*");
    Element e;
    Attr nsAttr;
    String attrpfx;
    String attrname;
    String attrval;
    NamedNodeMap nnm;
    int i, len;

    len = nl.getLength();

    for (int j=0; j < len; j++)
    {
        e = (Element) nl.item(j);
        System.out.println(e.getTagName() + ":");
        nnm = e.getAttributes();

        for (int k=0; k < nnm.getLength(); k++)
        {
            nsAttr = (Attr) nnm.item(k);
            attrpfx = nsAttr.getPrefix();
            attrname = nsAttr.getLocalName();
            attrval = nsAttr.getValue();
            System.out.println("   ATTRIBUTE Prefix Name :" + attrpfx + " Name:" + attrname + ":Value:" + attrval);
        }
    }
}

XML Parsing for Java  4-25
if (nnm != null)
{
    for (i=0; i < nnn.getLength(); i++)
    {
        nsAttr = (Attr) nnn.item(i);
        attrpfx = nsAttr.getPrefix();
        attrname = nsAttr.getLocalName();
        attrval = nsAttr.getNodeValue();
        System.out.println(" "+ attrpfx + ":" + attrname + ":
        + attrval);
    }
    System.out.println();
}

Performing DOM Operations with Events

The EventSample.java program shows how to register various events with an event
listener. For example, if a node is added to a specified DOM element, an event is
triggered, which causes the listener to print information about the event.

The program follows these steps:

1. Instantiate an event listener. When a registered change triggers an event, this event is passed to the event listener, which handles it. The following code fragment from EventSample.java shows the implementation of the listener:

   eventlistener evtlist = new eventlistener();
   ...
   class eventlistener implements EventListener
   {
       public eventlistener()
       {
           public void handleEvent(Event e)
           {
               String s = " Event "+e.getType()+" received "+"\n";
               s += " Event is cancelable :"+e.getCancelable()+"\n";
               s += " Event is bubbling event :"+e.getBubbles()+"\n";
               s += " The Target is "+ ((Node)(e.getTarget())).getNodeName()+"\n";
               System.out.println(s);
           }
       }
   }

2. Instantiate a new XMLDocument and then call getImplementation() to retrieve a DOMImplementation object. You can call the hasFeature() method to determine which features are supported by this implementation. The following code fragment from EventSample.java illustrates this technique:

   XMLDocument doc1 = new XMLDocument();
   DOMImplementation impl = doc1.getImplementation();
   System.out.println("The impl supports Events "+
   impl.hasFeature("Events", "2.0");
   System.out.println("The impl supports Mutation Events "+
   impl.hasFeature("MutationEvents", "2.0");

3. Register desired events with the listener. The following code fragment from EventSample.java registers three events on the document node:
doc1.addEventListener("DOMNodeRemoved", evtlist, false);
doc1.addEventListener("DOMNodeInserted", evtlist, false);
doc1.addEventListener("DOMCharacterDataModified", evtlist, false);

The following code fragment from EventSample.java creates a node of type
XMLElement and then registers three events on this node:
XMLElement el = (XMLElement)doc1.createElement("element");
...
el.addEventListener("DOMNodeRemoved", evtlist, false);
el.addEventListener("DOMNodeRemovedFromDocument", evtlist, false);
el.addEventListener("DOMCharacterDataModified", evtlist, false);
...

4. Perform actions that trigger events, which are then passed to the listener for
handling. The following code fragment from EventSample.java illustrates this
technique:
att.setNodeValue("abc");
el.appendChild(el1);
el.appendChild(text);
text.setNodeValue("xyz");
doc1.removeChild(el);

Performing DOM Operations with Ranges

According to the W3C DOM specification, a range identifies a range of content in a
Document, DocumentFragment, or Attr. It selects the content between a pair of
boundary-points that correspond to the start and the end of the range. Table 4–8
describes useful range methods accessible through XMLDocument.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cloneContents()</td>
<td>Duplicates the contents of a range</td>
</tr>
<tr>
<td>deleteContents()</td>
<td>Deletes the contents of a range</td>
</tr>
<tr>
<td>getCollapsed()</td>
<td>Returns TRUE is the range is collapsed</td>
</tr>
<tr>
<td>getEndContainer()</td>
<td>Obtains the node within which the range ends</td>
</tr>
<tr>
<td>getStartContainer()</td>
<td>Obtains the node within which the range begins</td>
</tr>
<tr>
<td>selectNode()</td>
<td>Selects a node and its contents</td>
</tr>
<tr>
<td>selectNodeContents()</td>
<td>Selects the contents within a node</td>
</tr>
<tr>
<td>setEnd()</td>
<td>Sets the attributes describing the end of a range</td>
</tr>
<tr>
<td>setStart()</td>
<td>Sets the attributes describing the beginning of a range</td>
</tr>
</tbody>
</table>

The DOMRangeSample.java program illustrates some of the things that you can do with
ranges.

The initial four steps of the "Performing Basic DOM Parsing” on page 4-15, from parser
creation to the getDocument() call, are the same as for DOMRangeSample.java. The
DOMRangeSample.java program then proceeds by following these steps:
1. After calling getDocument() to create the XMLDocument, create a range object with
createRange() and call setStart() and setEnd() to set its boundaries. The
following code fragment from DOMRangeSample.java illustrates this technique:

XMLDocument doc = parser.getDocument();
...  
Range r = (Range) doc.createRange();  
XMLNode c = (XMLNode) doc.getDocumentElement();  

// set the boundaries  
r.setStart(c,0);  
r.setEnd(c,1);  

2. Call XmlDocument methods to obtain information about the range and manipulate its contents. Table 4-8 describes useful methods. The following code fragment from DOMRangeSample.java selects the contents of the current node and prints it:

r.selectNodeContents(c);  
System.out.println(r.toString());  

The following code fragment clones a range contents and prints it:

XMLDocumentFragment df = (XMLDocumentFragment) r.cloneContents();  
df.print(System.out);  

The following code fragment obtains and prints the start and end containers for the range:

c = (XMLNode) r.getStartContainer();  
System.out.println(c.getText());  
c = (XMLNode) r.getEndContainer();  
System.out.println(c.getText());  

Only some of the features of the demo program are described in this section. For more detail, refer to the demo program itself.

Performing DOM Operations with TreeWalker

The W3C DOM Level 2 Traversal and Range specification defines the NodeFilter and TreeWalker interfaces. The XDK includes implementations of these interfaces.

A node filter is an object that can filter out certain types of Node objects. For example, it can filter out entity reference nodes but accept element and attribute nodes. You create a node filter by implementing the NodeFilter interface and then passing a Node object to the acceptNode() method. Typically, the acceptNode() method implementation calls getNodeType() to obtain the type of the node and compares it to static variables such as ELEMENT_TYPE, ATTRIBUTE_TYPE, and so forth, and then returns one of the static fields in Table 4-9 based on what it finds.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILTER_ACCEPT</td>
<td>Accept the node. Navigation methods defined for NodeIterator or TreeWalker will return this node.</td>
</tr>
<tr>
<td>FILTER_REJECT</td>
<td>Rejects the node. Navigation methods defined for NodeIterator or TreeWalker will not return this node. For TreeWalker, the children of this node will also be rejected. NodeIterators treat this as a synonym for FILTER_SKIP.</td>
</tr>
<tr>
<td>FILTER_SKIP</td>
<td>Skips this single node. Navigation methods defined for NodeIterator or TreeWalker will not return this node. For both NodeIterator and TreeWalker, the children of this node will still be considered.</td>
</tr>
</tbody>
</table>

You can use TreeWalker objects to traverse a document tree or subtree using the view of the document defined by their whatToShow flags and filters (if any). You can use the
XMLDocument.createTreeWalker() method to create a TreeWalker object by specifying the following:

- A root node for the tree
- A flag that governs the type of nodes it should include in the logical view
- A filter for filtering nodes
- A flag that determines whether entity references and their descendents should be included

Table 4–10 describes useful methods in the org.w3c.dom.traversal.TreeWalker interface.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstChild()</td>
<td>Moves the tree walker to the first visible child of the current node and returns the new node. If the current node has no visible children, then it returns null and retains the current node.</td>
</tr>
<tr>
<td>getRoot()</td>
<td>Obtains the root node of the tree walker as specified when it was created.</td>
</tr>
<tr>
<td>lastChild()</td>
<td>Moves the tree walker to the last visible child of the current node and returns the new node. If the current node has no visible children, then it returns null and retains the current node.</td>
</tr>
<tr>
<td>nextNode()</td>
<td>Moves the tree walker to the next visible node in document order relative to the current node and returns the new node.</td>
</tr>
</tbody>
</table>

The TreeWalkerSample.java program illustrates some of the things that you can do with node filters and tree traversals.

The initial four steps of the "Performing Basic DOM Parsing" on page 4-15, from parser creation to the getDocument() call, are the same as for TreeWalkerSample.java. The TreeWalkerSample.java program then proceeds by following these steps:

1. Create a node filter object. The acceptNode() method in the nf class, which implements the NodeFilter interface, invokes getNodeType() to obtain the type of node. The following code fragment from TreeWalkerSample.java illustrates this technique:

```java
NodeFilter n2 = new nf();
...

class nf implements NodeFilter
{
    public short acceptNode(Node node)
    {
        short type = node.getNodeType();

        if ((type == Node.ELEMENT_NODE) || (type == Node.ATTRIBUTE_NODE))
            return FILTER_ACCEPT;
        if ((type == Node.ENTITY_REFERENCE_NODE))
            return FILTER_REJECT;
        return FILTER_SKIP;
    }
}
```

2. Invoke the XMLDocument.createTreeWalker() method to create a tree walker. The following code fragment from TreeWalkerSample.java uses the root node of the XMLDocument as the root node of the tree walker and includes all nodes in the tree:
3. Obtain the root element of the TreeWalker object. The following code fragment illustrates this technique:

```java
XMLNode nn = (XMLNode)tw.getRoot();
```

4. Traverse the tree. The following code fragment illustrates how to walk the tree in document order by calling the `TreeWalker.nextNode()` method:

```java
while (nn != null) {
    System.out.println(nn.getNodeName() + " " + nn.getNodeValue());
    nn = (XMLNode)tw.nextNode();
}
```

The following code fragment illustrates how to walk the tree the left depth of the tree by calling the `firstChild()` method (you can traverse the right depth of the tree by calling the `lastChild()` method):

```java
while (nn != null) {
    System.out.println(nn.getNodeName() + " " + nn.getNodeValue());
    nn = (XMLNode)tw.firstChild();
}
```

Only some of the features of the demo program are described in this section. For more detail, refer to the demo program itself.

**Parsing XML with SAX**

SAX is a standard interface for event-based XML parsing. This section contains the following topics:

- Using the SAX API
- Performing Basic SAX Parsing
- Performing Basic SAX Parsing with Namespaces
- Performing SAX Parsing with XMLTokenizer

**Using the SAX API**

The SAX API, which is released in a Level 1 and Level 2 versions, is a set of interfaces and classes. We can divide the API into the following categories:

- Interfaces implemented by the Oracle XML parser.
- Interfaces that you must implement in your application. The SAX 2.0 interfaces are listed in Table 4–11.
Standard SAX classes.

Additional Java classes in org.xml.sax.helper. The SAX 2.0 helper classes are as follows:
- `AttributeImpl`, which makes a persistent copy of an `AttributeList`
- `DefaultHandler`, which is a base class with default implementations of the SAX2 handler interfaces listed in Table 4–11
- `LocatorImpl`, which makes a persistent snapshot of a Locator's values at specified point in the parse
- `NamespaceSupport`, which adds support for XML namespaces
- `XMLFilterImpl`, which is a base class used by applications that need to modify the stream of events
- `XMLReaderFactory`, which supports loading SAX parsers dynamically

Demonstration classes in the `nul` package.

Figure 4–5 illustrates how to create a SAX parser and use it to parse an input document.
The basic stages for parsing an input XML document with SAX are as follows:

1. Create a `SAXParser` object and configure its properties (see Table 4–5 for useful property methods). For example, set the validation mode of the parser.

2. Instantiate an event handler. The program should provide implementations of the handler interfaces in Table 4–11.

3. Register the event handlers with the parser. You must register your event handlers with the parser so that it knows which methods to invoke when a given event occurs. Table 4–12 lists registration methods available in `SAXParser`.

### Table 4–12  SAXParser Methods for Registering Event Handlers

<table>
<thead>
<tr>
<th>Method</th>
<th>Use this method to . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setContentHandler()</code></td>
<td>Register a content event handler with an application. The <code>org.xml.sax.DefaultHandler</code></td>
</tr>
<tr>
<td></td>
<td>class implements the <code>org.xml.sax.ContentHandler</code> interface. Applications can register</td>
</tr>
<tr>
<td></td>
<td>a new or different handler in the middle of a parse; the SAX parser must begin using</td>
</tr>
<tr>
<td></td>
<td>the new handler immediately.</td>
</tr>
<tr>
<td><code>setDTDHandler()</code></td>
<td>Register a DTD event handler. If the application does not register a DTD handler, all</td>
</tr>
<tr>
<td></td>
<td>DTD events reported by the SAX parser are silently ignored. Applications may register a</td>
</tr>
<tr>
<td></td>
<td>new or different handler in the middle of a parse; the SAX parser must begin using the</td>
</tr>
<tr>
<td></td>
<td>new handler immediately.</td>
</tr>
<tr>
<td><code>setErrorHandler()</code></td>
<td>Register an error event handler with an application. If the application does not register</td>
</tr>
<tr>
<td></td>
<td>an error handler, all error events reported by the SAX parser are silently ignored;</td>
</tr>
<tr>
<td></td>
<td>however, normal processing may not continue. It is highly recommended that all SAX</td>
</tr>
<tr>
<td></td>
<td>applications implement an error handler to avoid unexpected bugs. Applications may</td>
</tr>
<tr>
<td></td>
<td>register a new or different handler in the middle of a parse; the SAX parser must begin</td>
</tr>
<tr>
<td></td>
<td>using the new handler immediately.</td>
</tr>
<tr>
<td><code>setEntityResolver()</code></td>
<td>Register an entity resolver with an application. If the application does not register an</td>
</tr>
<tr>
<td></td>
<td>entity resolver, the <code>XMLReader</code> performs its own default resolution. Applications may</td>
</tr>
<tr>
<td></td>
<td>register a new or different resolver in the middle of a parse; the SAX parser must begin</td>
</tr>
<tr>
<td></td>
<td>using the new resolver immediately.</td>
</tr>
</tbody>
</table>
4. Parse the input document with the `SAXParser.parse()` method. All SAX interfaces are assumed to be synchronous: the parse method must not return until parsing is complete. Readers must wait for an event-handler callback to return before reporting the next event.

5. When the `SAXParser.parse()` method is called, the program invokes one of several callback methods implemented in the application. The methods are defined by the `ContentHandler`, `ErrorHandler`, `DTDHandler`, and `EntityResolver` interfaces implemented in the event handler. For example, the application can call the `startElement()` method when a start element is encountered.

Performing Basic SAX Parsing

The `SAXSample.java` program illustrates the basic steps of SAX parsing. The `SAXSample` class extends `HandlerBase`. The program receives an XML file as input, parses it, and prints information about the contents of the file.

The program follows these steps:

1. Store the Locator. The Locator associates a SAX event with a document location. The SAX parser provides location information to the application by passing a Locator instance to the `setDocumentLocator()` method in the content handler. The application can use the object to obtain the location of any other content handler event in the XML source document. The following code fragment from `SAXSample.java` illustrates this technique:

   ```java
   Locator locator;
   ```

2. Instantiate a new event handler. The following code fragment from `SAXSample.java` illustrates this technique:

   ```java
   SAXSample sample = new SAXSample();
   ```

3. Instantiate the SAX parser and configure it. The following code fragment from `SAXSample.java` sets the mode to DTD validation:

   ```java
   Parser parser = new SAXParser();
   ((SAXParser)parser).setValidationMode(SAXParser.DTD_VALIDATION);
   ```

4. Register event handlers with the SAX parser. You can use the registration methods in the `SAXParser` class, but you must implement the handler interfaces yourself. The following code fragment registers the handlers:

   ```java
   parser.setDocumentHandler(sample);
   parser.setEntityResolver(sample);
   parser.setDTDHandler(sample);
   parser.setErrorHandler(sample);
   ```

   The following code shows some of the `DocumentHandler` interface implementation:

   ```java
   public void setDocumentLocator (Locator locator) {
      System.out.println("SetDocumentLocator: ");
      this.locator = locator;
   }
   public void startDocument() {
      System.out.println("StartDocument");
   }
   public void endDocument() throws SAXException {
      System.out.println("EndDocument: ");
      ```
System.out.println("EndDocument");
}
public void startElement(String name, AttributeList atts)
    throws SAXException
{
    System.out.println("StartElement:"+name);
    for (int i=0;i<atts.getLength();i++)
    {
        String aname = atts.getName(i);
        String type = atts.getType(i);
        String value = atts.getValue(i);
        System.out.println("   "+aname+"("+type+")"+"="+value);
    }
}
...

The following code shows the EntityResolver interface implementation:

public InputSource resolveEntity (String publicId, String systemId)
    throws SAXException
{
    System.out.println("ResolveEntity:"+publicId+" "+systemId);
    System.out.println("Locator:"+locator.getPublicId() +" locator.getSystemId()+
    " +locator.getLineNumber() +" +locator.getColumnNumber());
    return null;
}

The following code shows the DTDHandler interface implementation:

public void notationDecl (String name, String publicId, String systemId)
{
    System.out.println("NotationDecl:"+name+" "+publicId+" "+systemId);
}
public void unparsedEntityDecl (String name, String publicId, String systemId, String notationName)
{
    System.out.println("UnparsedEntityDecl:"+name + " "+publicId+" "+
    systemId+" "+notationName);
}

The following code shows the ErrorHandler interface implementation:

public void warning (SAXParseException e)
    throws SAXException
{
    System.out.println("Warning:"+e.getMessage());
}
public void error (SAXParseException e)
    throws SAXException
{
    throw new SAXException(e.getMessage());
}
public void fatalError (SAXParseException e)
    throws SAXException
{
    System.out.println("Fatal error");
    throw new SAXException(e.getMessage());
}

5. Parse the input XML document. The following code fragment converts the
document to a URL and then parses it:
Performing Basic SAX Parsing with Namespaces

This section discusses the SAX2Namespace.java sample program, which implements an event handler named XMLDefaultHandler as a subclass of the org.xml.sax.helpers.DefaultHandler class. The easiest way to implement the ContentHandler interface is to extend the org.xml.sax.helpers.DefaultHandler class. The DefaultHandler class provides some default behavior for handling events, although typically the behavior is to do nothing.

The SAX2Namespace.java program overrides methods for only the events that it cares about. Specifically, the XMLDefaultHandler class implements only two methods: startElement() and endElement(). The startElement event is triggered whenever SAXParser encounters a new element within the XML document. When this event is triggered, the startElement() method prints the namespace information for the element.

The SAX2Namespace.java sample program follows these steps:

1. Instantiate a new event handler of type DefaultHandler. The following code fragment illustrates this technique:

   ```java
   DefaultHandler defHandler = new XMLDefaultHandler();
   ```

2. Create a SAX parser and set its validation mode. The following code fragment from SAXSample.java sets the mode to DTD validation:

   ```java
   Parser parser = new SAXParser();
   ((SAXParser)parser).setValidationMode(SAXParser.DTD_VALIDATION);
   ```

3. Register event handlers with the SAX parser. The following code fragment registers handlers for the input document, the DTD, entities, and errors:

   ```java
   parser.setContentHandler(defHandler);
   parser.setEntityResolver(defHandler);
   parser.setDTDHandler(defHandler);
   parser.setErrorHandler(defHandler);
   ```

The following code shows the XMLDefaultHandler implementation. The startElement() and endElement() methods print the qualified name, local name, and namespace URI for each element (refer to Table 4–7 for an explanation of these terms):

```java
class XMLDefaultHandler extends DefaultHandler {

    public void startElement(String uri, String localName, String qName, Attributes atts) throws SAXException {
        System.out.println("ELEMENT Qualified Name:" + qName);
        System.out.println("ELEMENT Local Name:" + localName);
        System.out.println("ELEMENT Namespace:" + uri);
        for (int i=0; i<atts.getLength(); i++) {
            qName = atts.getQName(i);
            localName = atts.getLocalName(i);
            uri = atts.getURI(i);
            System.out.println("ATTRIBUTE Qualified Name:" + qName);
            System.out.println("ATTRIBUTE Local Name:" + localName);
            System.out.println("ATTRIBUTE Namespace:" + uri);
        }
    }

    public void endElement(String uri, String localName, String qName) throws SAXException {
        System.out.println("EndElement qName:" + qName);
    }
}
```
System.out.println(" ATTRIBUTE Qualified Name: " + qName);
System.out.println(" ATTRIBUTE Local Name: " + localName);
System.out.println(" ATTRIBUTE Namespace: " + uri);

// You can get the type and value of the attributes either
// by index or by the Qualified Name.

String type = atts.getType(qName);
String value = atts.getValue(qName);

System.out.println(" ATTRIBUTE Type: " + type);
System.out.println(" ATTRIBUTE Value: " + value);

System.out.println();

}

public void endElement(String uri, String localName,
                      String qName) throws SAXException
{
  System.out.println("ELEMENT Qualified Name: " + qName);
  System.out.println("ELEMENT Local Name: " + localName);
  System.out.println("ELEMENT Namespace: " + uri);
}

4. Parse the input XML document. The following code fragment converts the
document to a URL and then parses it:

parser.parse(DemoUtil.createURL(argv[0]).toString());

Performing SAX Parsing with XMLTokenizer

You can create a simple SAX parser as an instance of the XMLTokenizer class and use the
parser to tokenize the input XML. Table 4–13 lists useful methods in the class.

Table 4–13 XMLTokenizer Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setToken()</td>
<td>Register a new token for XML tokenizer.</td>
</tr>
<tr>
<td>setErrorStream()</td>
<td>Register a output stream for errors</td>
</tr>
<tr>
<td>tokenize()</td>
<td>Tokenizes the input XML</td>
</tr>
</tbody>
</table>

SAX parsers with Tokenizer features must implement the XMLToken interface. The
callback method for XMLToken is token(), which receives an XML token and its
Corresponding value and performs an action. For example, you can implement
token() so that it prints the token name followed by the value of the token.

The Tokenizer.java program accepts an XML document as input, parses it, and prints
a list of the XML tokens. The program implements a doParse() method that does the
following:

1. Create a URL from the input XML stream:

   URL url = DemoUtil.createURL(arg);

2. Create an XMLTokenizer parser as follows:

   parser = new XMLTokenizer((XMLToken)new Tokenizer());
3. Register an output error stream as follows:
   
   ```java
   parser.setErrorStream (System.out);
   ```

4. Register tokens with the parser. The following code fragment from Tokenizer.java shows just some of the registered tokens:
   
   ```java
   parser.setToken (STagName, true);
   parser.setToken (EmptyElemTag, true);
   parser.setToken (STag, true);
   parser.setToken (ETag, true);
   parser.setToken (ETagName, true);
   ...
   ```

5. Tokenize the XML document as follows:
   
   ```java
   parser.tokenize (url);
   ```

   The `token()` callback method determines the action to take when an particular token is encountered. The following code fragment from Tokenizer.java shows some of the implementation of this method:
   
   ```java
   public void token (int token, String value)
   {
     switch (token)
     {
     case XMLToken.STag:
       System.out.println ('STag: ' + value);
       break;
     case XMLToken.ETag:
       System.out.println ('ETag: ' + value);
       break;
     case XMLToken.EmptyElemTag:
       System.out.println ('EmptyElemTag: ' + value);
       break;
     case XMLToken.AttValue:
       System.out.println ('AttValue: ' + value);
       break;
     ...
     default:
       break;
     }
   }
   ```

Parsing XML with JAXP

JAXP enables you to use the SAX and DOM parsers and the XSLT processor in your Java program. This section contains the following topics:

- Using the JAXP API
- Parsing with JAXP
- Performing Basic Transformations with JAXP

Using the JAXP API

The JAXP APIs, which are listed in Table 4-14, have an API structure consisting of abstract classes that provide a thin layer for parser pluggability. Oracle implemented JAXP based on the Sun Microsystems reference implementation.
Using the SAX API Through JAXP

You can rely on the factory design pattern to create new SAX parser engines with JAXP. Figure 4–6 illustrates the basic process.

**Figure 4–6  SAX Parsing with JAXP**

The basic steps for parsing with SAX through JAXP are as follows:

1. Create a new SAX parser factory with the `SAXParserFactory` class.
2. Configure the factory.
3. Create a new SAX parser (`SAXParser`) object from the factory.
4. Set the event handlers for the SAX parser.
5. Parse the input XML documents.

Using the DOM API Through JAXP

You can rely on the factory design pattern to create new DOM document builder engines with JAXP. Figure 4–7 illustrates the basic process.

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.xml.parsers</td>
<td>Provides standard APIs for DOM 2.0 and SAX 1.0 parsers. The package contains vendor-neutral factory classes that give you a SAXParser and a DocumentBuilder. DocumentBuilder creates a DOM-compliant Document object.</td>
</tr>
<tr>
<td>javax.xml.transform</td>
<td>Defines the generic APIs for processing XML transformation and performing a transformation from a source to a result.</td>
</tr>
<tr>
<td>javax.xml.transform.dom</td>
<td>Provides DOM-specific transformation APIs.</td>
</tr>
<tr>
<td>javax.xml.transform.sax</td>
<td>Provides SAX2-specific transformation APIs.</td>
</tr>
<tr>
<td>javax.xml.transform.stream</td>
<td>Provides stream- and URI-specific transformation APIs.</td>
</tr>
</tbody>
</table>
The basic steps for parsing with DOM through JAXP are as follows:
1. Create a new DOM parser factory with the `DocumentBuilderFactory` class.
2. Configure the factory.
3. Create a new DOM builder (`DocumentBuilder`) object from the factory.
4. Set the error handler and entity resolver for the DOM builder.
5. Parse the input XML documents.

Transforming XML Through JAXP
The basic steps for transforming XML through JAXP are as follows:
1. Create a new transformer factory. Use the `TransformerFactory` class.
2. Configure the factory.
3. Create a new transformer from the factory and specify an XSLT stylesheet.
4. Configure the transformer.
5. Transform the document.

Parsing with JAXP
The `JAXPExamples.java` program illustrates the basic steps of parsing with JAXP. The program implements the following methods and uses them to parse and perform additional processing on XML files in the `/jaxp` directory:

- `basic()`
- `identity()`
- `namespaceURI()`
- `templatesHandler()`
- `contentHandler2contentHandler()`
- `contentHandler2DOM()`
- `reader()`
- `xmlFilter()`
- `xmlFilterChain()`
The program creates URLs for the jaxpone.xml and jaxpone.xsl sample XML files and then calls the preceding methods in sequence. The basic design of the demo is as follows (to save space only the basic() method is shown):

```java
public class JAXPExamples
{
    public static void main(String argv[])
        throws TransformerException, TransformerConfigurationException,
        IOException, SAXException, ParserConfigurationException,
        FileNotFoundException
    {
        try {
            URL xmlURL = createURL("jaxpone.xml");
            String xmlID = xmlURL.toString();
            URL xslURL = createURL("jaxpone.xsl");
            String xslID = xslURL.toString();
            //
            System.out.println("--- basic ---");
            basic(xmlID, xslID);
            System.out.println();
            ...
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
    //
    public static void basic(String xmlID, String xslID)
        throws TransformerException, TransformerConfigurationException
    {
        TransformerFactory tfactory = TransformerFactory.newInstance();
        Transformer transformer = tfactory.newTransformer(new StreamSource(xslID));
        StreamSource source = new StreamSource(xmlID);
        transformer.transform(source, new StreamResult(System.out));
        ...
    }
}
```

The reader() method in JAXPExamples.java program shows a simple technique for parsing an XML document with SAX. It follows these steps:

1. **Create a new instance of a TransformerFactory and then cast it to a SAXTransformerFactory.** The application can use the SAX factory to configure and obtain SAX parser instances. For example:

   ```java
   TransformerFactory tfactory = TransformerFactory.newInstance();
   SAXTransformerFactory stfactory = (SAXTransformerFactory)tfactory;
   ```

2. **Create an XML reader by creating a StreamSource object from a stylesheet and passing it to the factory method newXMLFilter().** This method returns an XMLFilter object that uses the specified Source as the transformation instructions. For example:

   ```java
   URL xslURL = createURL("jaxpone.xsl");
   String xslID = xslURL.toString();
   ...
   StreamSource streamSource = new StreamSource(xslID);
   XMLReader reader = stfactory.newXMLFilter(streamSource);
   ```

3. **Create content handler and register it with the XML reader.** The following example creates an instance of the class oraContentHandler, which is created by compiling the oraContentHandler.java program in the demo directory:
ContentHandler contentHandler = new oraContentHandler();
reader.setContentHandler(contentHandler);

The following code fragment shows some of the implementation of the oraContentHandler class:

class oraContentHandler implements ContentHandler
{
    private static final String TRADE_MARK = "Oracle 9i ";

    public void setDocumentLocator(Locator locator)
    {
        System.out.println(TRADE_MARK + "- setDocumentLocator");
    }

    public void startDocument()
        throws SAXException
    {
        System.out.println(TRADE_MARK + "- startDocument");
    }

    public void endDocument()
        throws SAXException
    {
        System.out.println(TRADE_MARK + "- endDocument");
    }
    ...

4. Parse the input XML document by passing the InputSource to the XMLReader.parse() method. For example:

    InputSource is = new InputSource(xmlID);
    reader.parse(is);

Performing Basic Transformations with JAXP

You can use JAXP to transform any class of the interface Source into a class of the interface Result. Table 4–15 shows some sample transformations.

<table>
<thead>
<tr>
<th>Use JAXP to transform this class . . .</th>
<th>Into this class . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMSource</td>
<td>DOMResult</td>
</tr>
<tr>
<td>StreamSource</td>
<td>StreamResult</td>
</tr>
<tr>
<td>SAXSource</td>
<td>SAXResult</td>
</tr>
</tbody>
</table>

These transformations accept the following types of input:

- XML documents
- stylesheets
- The ContentHandler class defined in oraContentHandler.java

For example, you can use the identity() method to perform a transformation in which the output XML document is the same as the input XML document. You can use the xmlFilterChain() method to apply three stylesheets in a chain.

The basic() method shows how to perform a basic XSLT transformation. The method follows these steps:
1. Create a new instance of a TransformerFactory. For example:

   TransformerFactory tfactory = TransformerFactory.newInstance();

2. Create a new XSL transformer from the factory and specify the stylesheet to use for the transformation. The following example specifies the jaxpone.xsl stylesheet:

   URL xslURL = createURL("jaxpone.xsl");
   String xslID = xslURL.toString();
   . . .
   Transformer transformer = tfactory.newTransformer(new StreamSource(xslID));

3. Set the stream source to the input XML document. The following fragment from the basic() method sets the stream source to jaxpone.xml:

   URL xmlURL = createURL("jaxpone.xml");
   String xmlID = xmlURL.toString();
   . . .
   StreamSource source = new StreamSource(xmlID);

4. Transform the document from a StreamSource to a StreamResult. The following example transforms a StreamSource into a StreamResult:

   transformer.transform(source, new StreamResult(System.out));

Compressing XML

The Oracle XDK enables you to use SAX or DOM to parse XML and then write the parsed data to a compressed binary stream. You can then reverse the process and reconstruct the XML data. This section contains the following topics:

- Compressing and Decompressing XML from DOM
- Compressing and Decompressing XML from SAX

Compressing and Decompressing XML from DOM

The DOMCompression.java and DOMDeCompression.java programs illustrate the basic steps of DOM compression and decompression. The most important DOM compression methods are the following:

- XMLDocument.writeExternal() saves the state of the object by creating a binary compressed stream with information about the object.
- XMLDocument.readExternal() reads the information written in the compressed stream by the writeExternal() method and restores the object.

Compressing a DOM Object

The basic technique for serialization is create an XMLDocument by parsing an XML document, initialize an ObjectOutputStream, and then call XMLDocument.writeExternal() to write the compressed stream.

The DOMCompression.java program follows these steps:

1. Create a DOM parser, parse an input XML document, and obtain the DOM representation. This technique is described in "Performing Basic DOM Parsing" on page 4-15. The following code fragment from DOMCompression.java illustrates this technique:

   public class DOMCompression
```java
{  
    static OutputStream out = System.out;
    public static void main(String[] args)
    {
        XMLDocument doc = new XMLDocument();
        DOMParser parser = new DOMParser();
        try
        {
            parser.setValidationMode(XMLParser.SCHEMA_VALIDATION);
            parser.setPreserveWhitespace(false);
            parser.retainCDATASection(true);
            parser.parse(createURL(args[0]));
            doc = parser.getDocument();
...  
        
        2. Create a FileOutputStream and wrap it in an ObjectOutputStream for serialization. The following code fragment creates the xml.ser output file:

        OutputStream os = new FileOutputStream("xml.ser");
        ObjectOutputStream oos = new ObjectOutputStream(os);

        3. Serialize the object to the file by calling XMLDocument.writeExternal(). This method saves the state of the object by creating a binary compressed stream with information about this object. The following statement illustrates this technique:

        doc.writeExternal(oos);

    Decompressing a DOM Object

    The basic technique for decompression is to create an ObjectInputStream object and then call XMLDocument.readExternal() to read the compressed stream. The DOMDeCompression.java program follows these steps:

    1. Create a file input stream for the compressed file and wrap it in an ObjectInputStream. The following code fragment from DOMDeCompression.java creates a FileInputStream from the compressed file created in the previous section:

        InputStreamReader is;
        ObjectInputStream ois;
        ...
        is = new FileInputStream("xml.ser");
        ois = new ObjectInputStream(is);

    2. Create a new XML document object to contain the decompressed data. The following code fragment illustrates this technique:

        XMLDocument serializedDoc = null;
        serializedDoc = new XMLDocument();

    3. Read the compressed file by calling XMLDocument.readExternal(). The following code fragment read the data and prints it to System.out:

        serializedDoc.readExternal(ois);
        serializedDoc.print(System.out);

    Compressing and Decompressing XML from SAX

    The SAXCompression.java program illustrates the basic steps of parsing a file with SAX, writing the compressed stream to a file, and then reading the serialized data from the file. The important classes are as follows:
```
Compressing XML

- **CXMLHandlerBase** is a SAX Handler that compresses XML data based on SAX events. To use the SAX compression, implement this interface and register with the SAX parser by calling `Parser.setDocumentHandler()`.

- **CXMLParser** is an XML parser that regenerates SAX events from a compressed stream.

### Compressing a SAX Object

The basic technique for serialization is to register a `CXMLHandlerBase` handler with a SAX parser, initialize an `ObjectOutputStream`, and then parse the input XML. The `SAXCompression.java` program follows these steps:

1. Create a `FileOutputStream` and wrap it in an `ObjectOutputStream`. The following code fragment from `SAXCompression.java` creates the xml.ser file:
   ```java
   String compFile = "xml.ser";
   FileOutputStream outStream = new FileOutputStream(compFile);
   ObjectOutputStream out = new ObjectOutputStream(outStream);
   ```

2. Create the SAX event handler. The `CXMLHandlerBase` class implements the `ContentHandler`, `DTDHandler`, `EntityResolver`, and `ErrorHandler` interfaces. The following code fragment illustrates this technique:
   ```java
   CXMLHandlerBase cxml = new CXMLHandlerBase(out);
   ```

3. Create the SAX parser. The following code fragment illustrates this technique:
   ```java
   SAXParser parser = new SAXParser();
   ```

4. Configure the SAX parser. The following code fragment sets the content handler and entity resolver, and also sets the validation mode:
   ```java
   parser.setContentHandler(cxml);
   parser.setEntityResolver(cxml);
   parser.setValidationMode(XMLConstants.NONVALIDATING);
   ```

   Note that `oracle.xml.comp.CXMLHandlerBase` implements both `DocumentHandler` and `ContentHandler` interfaces, but use of the SAX 2.0 `ContentHandler` interface is preferred.

5. Parse the XML. The program writes the serialized data to the `ObjectOutputStream`. The following code fragment illustrates this technique:
   ```java
   parser.parse(url);
   ```

### Decompressing a SAX Object

The basic technique for deserialization of a SAX object is to create a SAX compression parser with the `CXMLParser` class, set the content handler for the parser, and then parse the compressed stream.

The `SAXDeCompression.java` program follows these steps:

1. Create a SAX event handler. The `SampleSAXHandler.java` program creates a handler for use by `SAXDeCompression.java`. The following code fragment from `SAXDeCompression.java` creates handler object:
   ```java
   SampleSAXHandler xmlHandler = new SampleSAXHandler();
   ```

2. Create the SAX parser by instantiating the `CXMLParser` class. This class implements the regeneration of XML documents from a compressed stream by generating SAX events from them. The following code fragment illustrates this technique:
3. Set the event handler for the SAX parser. The following code fragment illustrates this technique:
   
   ```java
   parser.setContentHandler(xmlHandler);
   ```

4. Parse the compressed stream and generates the SAX events. The following code receives a filename from the command line and parses the XML:

   ```java
   parser.parse(args[0]);
   ```

**Tips and Techniques for Parsing XML**

This section contains the following topics:

- Extracting Node Values from a DOM Tree
- Merging Documents with `appendChild()`
- Parsing DTDs
- Handling Character Sets with the XML Parser

**Extracting Node Values from a DOM Tree**

You can use the `selectNodes()` method in the `XMLNode` class to extract content from a DOM tree or subtree based on the select patterns allowed by XSL. You can use the optional second parameter of `selectNodes()` to resolve namespace prefixes, that is, to return the expanded namespace URL when given a prefix. The `XMLElement` class implements `NSResolver`, so a reference to an `XMLElement` object can be sent as the second parameter. `XMLElement` resolves the prefixes based on the input document. You can use the `NSResolver` interface if you need to override the namespace definitions.

The sample code in Example 4–4 illustrates how to use `selectNodes()`.

**Example 4–4  Extracting Contents of a DOM Tree with selectNodes()**

```java
// // selectNodesTest.java
// import java.io.*;
import oracle.xml.parser.v2.*;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
public class selectNodesTest
{
    public static void main(String[] args)
        throws Exception
    {
        // supply an xpath expression
        String pattern = "//family/member/text()";
        // accept a filename on the command line
        // run the program with $ORACLE_HOME/xdk/demo/java/parser/common/family.xml
        String file    = args[0];
        if (args.length == 2)
            pattern = args[1];
```
DOMParser dp = new DOMParser();

dp.parse(DemoUtil.createURL(file)); // include createURL from DemoUtil
XMLDocument xd = dp.getDocument();
XMLElement element = (XMLElement) xd.getDocumentElement();
NodeList nl = element.selectNodes(pattern, element);
for (int i = 0; i < nl.getLength(); i++)
{
    System.out.println(nl.item(i).getNodeValue());
} // end for
} // end main
} // end selectNodesTest

To test the program, create a file with the code in Example 4–4 and then compile it in
the $ORACLE_HOME/xdk/demo/java/parser/common directory. Pass the filename
family.xml to the program as a parameter to traverse the <family> tree. The output
should be as follows:

% java selectNodesTest family.xml
Sarah
Bob
Joanne
Jim

Now run the following to determine the values of the memberid attributes of all
<member> elements in the document:

% java selectNodesTest family.xml //member/@memberid
m1
m2
m3
m4

Merging Documents with appendChild()

Suppose that you want to write a program so that a user can fill in a client-side Java
form and obtain an XML document. Suppose that your Java program contains the
following variables of type String:
String firstname = 'Gianfranco';
String lastname = 'Pietraforte';

You can use either of the following techniques to insert this information into an XML
document:

- Create an XML document in a string and then parse it. For example:

  String xml = "<person><first>"+firstname+"</first>"+
  "<last>"+lastname+"</last></person>";
  DOMParser d = new DOMParser();
  d.parse(new StringReader(xml));
  Document xmldoc = d.getDocument();

- Use DOM APIs to construct an XML document, creating elements and then
  appending them to one another. For example:

  Document xmldoc = new XMLDocument();
  Element e1 = xmldoc.createElement("person");
  xmldoc.appendChild(e1);
  Element e2 = xmldoc.createElement("firstname");
Tips and Techniques for Parsing XML

**XML Parsing for Java**

```java
e1.appendChild(e2);
Text t = xmldoc.createText('Larry');
e2.appendChild(t);
```

Note that you can only use the second technique on a single DOM tree. For example, suppose that you write the code snippet in Example 4–5.

**Example 4–5  Incorrect Use of appendChild()**

```java
XMLDocument xmldoc1 = new XMLDocument();
XMLElement e1 = xmldoc1.createElement("person");
XMLDocument xmldoc2 = new XMLDocument();
XMLElement e2 = xmldoc2.createElement("firstname");
e1.appendChild(e2);
```

The preceding code raises a DOM exception of `WRONG_DOCUMENT_ERR` when calling `XMLElement.appendChild()` because the owner document of `e1` is `xmldoc1` whereas the owner of `e2` is `xmldoc2`. The `appendChild()` method only works within a single tree, but the code in Example 4–5 uses two different trees.

You can use the `XMLDocument.importNode()` method, introduced in DOM 2, and the `XMLDocument.adoptNode()` method, introduced in DOM 3, to copy and paste a DOM document fragment or a DOM node across different XML documents. The commented lines in Example 4–6 show how to perform this task.

**Example 4–6  Merging Documents with appendChild**

```java
XMLDocument doc1 = new XMLDocument();
XMLElement element1 = doc1.createElement("person");
XMLDocument doc2 = new XMLDocument();
XMLElement element2 = doc2.createElement("firstname");
// element2 = doc1.importNode(element2);
// element2 = doc1.adoptNode(element2);
element1.appendChild(element2);
```

**Parsing DTDs**

This section discusses techniques for parsing DTDs. It contains the sections:

- **Loading External DTDs**
- **Caching DTDs with setDoctype**

**Loading External DTDs**

If you call the `DOMParser.parse()` method to parse the XML Document as an `InputStream`, then use the `DOMParser.setBaseURL()` method to recognize external DTDs within your Java program. This method points to a location where the DTDs are exposed.

The following procedure describes how to load and parse a DTD:

1. Load the DTD as an `InputStream`. For example, assume that you want to validate documents against the `/mydir/my.dtd` external DTD. You can use the following code:
   ```java
   InputStream is = MyClass.class.getResourceAsStream("/mydir/my.dtd");
   ```
   This code opens `./mydir/my.dtd` in the first relative location in the `CLASSPATH` where it can be found, including the JAR file if it is in the `CLASSPATH`.

2. Create a DOM parser and set the validation mode. For example, use this code:
DOMParser d = new DOMParser();
d.setValidationMode(DTD_VALIDATION);

3. Parse the DTD. The following example passes the InputStream object to the
   DOMParser.parseDTD() method:
   d.parseDTD(is, "rootelementname");

4. Get the document type and then set it. The getDoctype() method obtains the DTD
   object and the setDoctype() method sets the DTD to use for parsing. The
   following example illustrates this technique:
   d.setDoctype(d.getDoctype());

   The following code demonstrates an alternative technique. You can invoke the
   parseDTD() method to parse a DTD file separately and get a DTD object:
   d.parseDTD(new FileReader("/mydir/my.dtd");
   DTD dtd = d.getDoctype();
   parser.setDoctype(dtd);

5. Parse the input XML document. For example, the following code parses
   mydoc.xml:
   d.parse("mydoc.xml");

Caching DTDs with setDoctype
The XML parser for Java provides for DTD caching in validation and nonvalidation
modes through the DOMParser.setDoctype() method. After you set the DTD with this
method, the parser caches this DTD for further parsing. Note that DTD caching is
optional and is not enabled automatically.
Assume that your program must parse several XML documents with the same DTD.
After you parse the first XML document, you can obtain the DTD from the parser and
set it as in the following example:
DOMParser parser = new DOMParser();
DTD dtd = parser.getDoctype();
parser.setDoctype(dtd);

The parser caches this DTD and uses it for parsing subsequent XML documents.
Example 4–7 provides a more complete illustration of how you can invoke
DOMParser.setDoctype() to cache the DTD.

Example 4–7  DTDSample.java
/**
 * DESCRIPTION
 * This program illustrates DTD caching.
 */
import java.net.URL;
import java.io.*;
import org.xml.sax.InputSource;
import oracle.xml.parser.v2.*;

public class DTDSample
{
    static public void main(String[] args)
    {
        try

Oracle XML Developer’s Kit Programmer’s Guide
if (args.length != 3)
{
    System.err.println("Usage: java DTDSample dtd rootelement xmldoc");
    System.exit(1);
}

// Create a DOM parser
DOMParser parser = new DOMParser();

// Configure the parser
parser.setErrorStream(System.out);
parser.showWarnings(true);

// Create a FileReader for the DTD file specified on the command
// line and wrap it in an InputSource
FileReader r = new FileReader(args[0]);
InputSource inSource = new InputSource(r);

// Create a URL from the command-line argument and use it to set the
// system identifier
inSource.setSystemId(DemoUtil.createURL(args[0]).toString());

// Parse the external DTD from the input source. The second argument is
// the name of the root element.
parser.parseDTD(inSource, args[1]);
DTD dtd = parser.getDoctype();

// Create a FileReader object from the XML document specified on the
// command line
r = new FileReader(args[2]);

// Wrap the FileReader in an InputSource, create a URL from the filename,
// and set the system identifier
inSource = new InputSource(r);
inSource.setSystemId(DemoUtil.createURL(args[2]).toString());

// ****************
parser.setDoctype(dtd);
// ****************

parser.setValidationMode(DOMParser.DTD_VALIDATION);
// parser.setAttribute(DOMParser.USE_DTD_ONLY_FOR_VALIDATION,Boolean.TRUE);
parser.parse(inSource);

// Obtain the DOM tree and print
XMLDocument doc = parser.getDocument();
doc.print(new PrintWriter(System.out));

} catch (Exception e)
{
    System.out.println(e.toString());
}

If the cached DTD Object is used only for validation, then set the DOMParser.USE_DTD_ ONLY_FOR_VALIDATION attribute. Otherwise, the XML parser will copy the DTD object and add it to the resulting DOM tree. You can set the parser as follows:
parser.setAttribute(DOMParser.USE_DTD_ONLY_FOR_VALIDATION, Boolean.TRUE);

### Handling Character Sets with the XML Parser

This section contains the following topics:

- Detecting the Encoding of an XML File on the Operating System
- Detecting the Encoding of XML Stored in an NCLOB Column
- Writing an XML File in a Nondefault Encoding
- Working with XML in Strings
- Parsing XML Documents with Accented Characters
- Handling Special Characters in Tag Names

#### Detecting the Encoding of an XML File on the Operating System

When reading an XML file stored on the operating system, do not use the `FileReader` class. Instead, use the XML parser to detect the character encoding of the document automatically. Given a binary `FileInputStream` with no external encoding information, the parser automatically determines the character encoding based on the byte-order mark and encoding declaration of the XML document. You can parse any well-formed document in any supported encoding with the sample code in the `AutoDetectEncoding.java` demo. This demo is located in `$ORACLE_HOME/xdk/demo/java/parser/dom`.

**Note:** Include the proper encoding declaration in your document according to the specification. `setEncoding()` cannot set the encoding for your input document. Rather, it is used with `oracle.xml.parser.v2.XMLDocument` to set the correct encoding for printing.

#### Detecting the Encoding of XML Stored in an NCLOB Column

Suppose that you load XML into an NCLOB column of a database using UTF-8 encoding. The XML contains two UTF-8 multibyte characters:

G(0xc2,0x82)otingen, Br(0xc3,0xbc)ck_W

You write a Java stored function that does the following:

1. Uses the default connection object to connect to the database.
2. Runs a `SELECT` query.
3. Obtains the `oracle.jdbc.OracleResultSet` object.
4. Calls the `OracleResultSet.getCLOB()` method.
5. Calls the `getAsciiStream()` method on the CLOB object.
6. Executes the following code to get the XML into a DOM object:

   ```java
   DOMParser parser = new DOMParser();
   parser.setPreserveWhitespace(true);
   parser.parse(istr);
   // istr getAsciiStream XMLDocument xmldoc = parser.getDocument();
   ```

   The program throws an exception stating that the XML contains an invalid UTF-8 encoding even though the character (0xc2, 0x82) is valid UTF-8. The problem is that
the character can be distorted when the program calls the
OracleResultSet.getAsciiStream() method. To solve this problem, invoke
the getUnicodeStream() and getBinaryStream() methods instead of
getAsciiStream(). If this technique does not work, then try to print the
characters to make sure that they
are not distorted before they are sent to the parser in when you call
DOMParser.parse(istr).

**Writing an XML File in a Nondefault Encoding**

You should not use the FileWriter class when writing XML files because it depends
on the default character encoding of the runtime environment. The output file can
suffer from a parsing error or data loss if the document contains characters that are not
available in the default character encoding.

UTF-8 encoding is popular for XML documents, but UTF-8 is not usually the default
file encoding of Java. Using a Java class in your program that assumes the default file
encoding can cause problems. To avoid these problems, you can use the technique
illustrated in the I18nSafeXMLFileWritingSample.java program in $ORACLE_HOME/xdk/demo/java/parser/dom.

Note that you cannot use System.out.println() to output special characters. You
need to use a binary output stream such as OutputStreamWriter that is encoding
aware. You can construct an OutputStreamWriter and use the write(char[],int,int)
method to print, as in the following example:

```java
/* Java encoding string for ISO8859-1*/
OutputStreamWriter out = new OutputStreamWriter(System.out, "8859_1");
OutputStreamWriter.write(...);
```

**Working with XML in Strings**

Currently, there is no method that can directly parse an XML document contained in a
String. You need to convert the string into an InputStream or InputSource object
before parsing.

One technique is to create aByteArrayInputStream that uses the bytes in the string.
For example, assume that xmlDoc is a reference to a string of XML. You can use
technique shown in Example 4–8 to convert the string to a byte array, convert the array
to a ByteArrayInputStream, and then parse.

**Example 4–8 Converting XML in a String**

```java
// create parser
DOMParser parser=new DOMParser();
// create XML document in a string
String xmlDoc =
    '<xml version='1.0'>/*
    '<hello>'+
     ' <world/>'+
    '</hello>'; // convert string to bytes to stream
byte aByteArr [] = xmlDoc.getBytes();
ByteArrayInputStream bais = new ByteArrayInputStream(aByteArr,0,aByteArr.length);
// parse and obtain DOM tree
DOMParser.parse(bais);
XMLDocument doc = parser.getDocument();

Suppose that you want to convert the XMLDocument object created in the previous code
back to a string. You can perform this task by wrapping a StringWriter in a
PrintWriter. The following example illustrates this technique:
StringWriter sw = new StringWriter();
PrintWriter pw = new PrintWriter(sw);
doc.print(pw);
String YourDocInString = sw.toString();

ParseXMLFromString.java, which is located in $ORACLE_HOME/xdk/demo/java/parser/dom, is a complete program that creates an XML document as a string and parses it.

Parsing XML Documents with Accented Characters

Assume that an input XML file contains accented characters such as an é. Example 4-9 shows one way to parse an XML document with accented characters.

Example 4-9 Parsing a Document with Accented Characters

```java
DOMParser parser = new DOMParser();
parser.setPreserveWhitespace(true);
parser.setErrorStream(System.err);
parser.setValidationMode(false);
parser.showWarnings(true);
parser.parse (new FileInputStream(new File("file_with_accents.xml")));
```

When you attempt to parse the XML file, the parser can sometimes throw an “Invalid UTF-8 encoding” exception. If you explicitly set the encoding to UTF-8, or if you do not specify it at all, then the parser interprets an accented character—which has an ASCII value greater than 127—as the first byte of a UTF-8 multibyte sequence. If the subsequent bytes do not form a valid UTF-8 sequence, then you receive an error.

This error means that your XML editor did not save the file with UTF-8 encoding. For example, it may have saved it with ISO-8859-1 encoding. The encoding is a particular scheme used to write the Unicode character number representation to disk. Adding the following element to the top of an XML document does not itself cause your editor to write out the bytes representing the file to disk with UTF-8 encoding:

```xml
<?xml version="1.0" encoding="UTF-8"?>
```

One solution is to read in accented characters in their hex or decimal format within the XML document, for example, &\#xd9; If you prefer not to use this technique, however, then you can set the encoding based on the character set that you were using when you created the XML file. For example, try setting the encoding to ISO-8859-1 (Western European ASCII) or to something different, depending on the tool or operating system you are using.

Handling Special Characters in Tag Names

Special characters such as &, $, and #, and so on are not legal in tag names. For example, if a document names tags after companies, and if the document includes the tag `<A&B>`, then the parser issues an error about invalid characters.

If you are creating an XML document from scratch, then you can work around this problem by using only valid NameChars. For example, you can name the tag `<A_B>`, `<AB>`, `<A_AND_B>` and so on. If you are generating XML from external data sources such as database tables, however, then XML 1.0 does not address this problem.

The datatype XMLType addresses this problem by providing the setConvertSpecialChars and convert functions in the DBMS_XMLGEN package. You can use these functions to control the use of special characters in SQL names and XML names. The SQL to XML name mapping functions escape invalid XML NameChar characters in the format of _XHHHH_, where _HHHH_ is the Unicode value of the invalid
character. For example, table name V$SESSION is mapped to XML name V_X0024_SESSION.

Escaping invalid characters is another workaround to give users a way to serialize names so that they can reload them somewhere else.
This chapter contains these topics:

- Introduction to Binary XML for Java
- Models for Using Binary XML
- The Parts of Binary XML for Java
- Binary XML Vocabulary Management
- Using Java Binary XML Package

Introduction to Binary XML for Java

Binary XML was introduced in Oracle 11g Release 1 (11.1). Binary XML makes it possible to encode and decode between XML text and compressed binary XML. For efficiency, the DOM and SAX APIs are provided on top of binary XML for direct consumption by the XML applications. Compression and decompression of fragments of an XML document facilitate incremental processing.

This chapter assumes that you are familiar with the XML Parser for Java.

See Also: Chapter 4, "XML Parsing for Java"

Binary XML Storage Format

An XMLType storage option is provided to enable storing XML documents in the new binary format. The new storage option is in addition to the existing CLOB and object-relational storage options. XMLType tables and columns can be created using the new binary XML storage option. The XML data in binary format can be accessed and manipulated by all the existing SQL operators and functions and PL/SQL APIs that operate on XMLType.

Binary XML is a compact XML-schema-aware encoding of XML data, but it can be used with XML data that is not based on an XML schema. You can also use binary XML for XML data which is outside the database (in a client-side application, for instance). Binary XML allows for encoding and decoding of XML documents, from text to binary and binary to text. Binary XML is post-parse persistent XML with native database datatypes.

Binary XML provides more efficient database storage, updating, indexing, query performance, and fragment extraction than unstructured storage. It can store data and metadata together or separately.
Binary XML Processors

A binary XML processor is an abstract term for describing a component that processes and transforms binary XML format into text and XML text into binary XML format. It can also provide a cache for storing schemas. The base class for a binary XML processor is `BinXMLProcessor`. A binary XML processor can originate or receive network protocol requests.

Models for Using Binary XML

There are several models for using binary XML in applications. First, here is a glossary of terms:

Glossary for Binary XML

Here is a glossary of terms for binary XML usage:

- **doc-id**: Each encoded XML document is identified by a unique doc-id. It is either a 16-byte Global User ID (GUID) or an opaque sequence of bytes like a URL.
- **token table**: When a text XML document does not have a schema associated with it, then a token (or symbol) table is used to minimize space for repeated items.
- **vocabulary id**: Can be a schema-id or a namespace URI identification for a token table.
- **schema-id**: A unique opaque binary identifier for a schema scoped to the binary XML processor. The schema-id is unique for a binary XML processor and is identifiable only within the scope of that binary XML processor. The schema-id remains constant even when the schema is evolved. A schema-id represents the entire set of schema documents, including imported and included schemas.
- **schema version**: Every annotated schema has a version number associated with it. The version number is specified as part of the system level annotations. It is incremented by the binary XML processor when a schema is evolved (that is, a new version of the same schema is registered with the binary XML processor).
- **partial validity**: Binary XML stream encoding using schema implies at least partial validity with respect to the schema. Partial validity implies no validation for unique keys, keyrefs, IDs, or IDREFs.

Standalone Model

This is the simplest usage scenario for binary XML. There is a single binary XML processor. The only repository available is the local in-memory vocabulary cache that is not persistent and is only available for the life of the binary XML processor. All schemas must be registered in advance with the binary XML Processor before the encoding, or can be registered automatically when the XML Processor sees the `xsi:SchemaLocation` tag. For decoding, the schema is already available in the vocabulary cache.

If the decoding occurs in a different binary XML processor, see the different Web Services models described here.

Client-Server Model

In this scenario, the binary XML processor is connected to a database using JDBC. It is assumed that the schema is registered with the database before encoding.
Here is an example of how to achieve that:

BEGIN
  DBMS_XMLSCHEMA.registerSchema{
    SCHEMAURL =>
      'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    SCHEMADOC =>
      bfilename('XMLDIR','purchaseOrder.xsd'),
    CSID => nls_charset_id('AL32UTF8'),
    GENTYPES => FALSE,
    OPTIONS => REGISTER_BINARYXML
  };
END;
/

Unless a separate connection is specified for data (using associateDataConnection()) it is assumed that all data and metadata is stored and retrieved using a single connection for encoding and decoding.

Web Services Model with Repository

In this scenario there are multiple clients, each running a binary XML processor. One client does the encoding and the other client does the decoding. There is a common repository (that is not necessarily a database) connected to all the clients for metadata storage. It can be a file system or some other repository. The first binary XML processor ensures that the schema is registered with the repository before performing the encoding, or the schema might be automatically registered using the xsi:schemaLocation tag at the time of encoding. The second binary XML processor is used for decoding, is not aware of the location of the schema, and fetches the schema from the repository.

If the first binary XML processor registers a schema and the second binary XML processor registers the same schema in the repository, the binary XML processor does not compile the schema, but simply returns the vocabulary-id of the existing compiled schema in the local vocabulary cache.

The BinXMLProcessor is not thread-safe, so multiple threads or clients accessing the repository need to implement their own thread safety scheme.

Web Services Model Without Repository

In this scenario, there are multiple clients, each running a binary XML processor. Encoding and decoding can happen on different clients. There is no common metadata repository. The encoder has to ensure that the binary data passed to the next client is independent of schema: that is, has inline token definitions. This can be achieved by setting schemaAware = false and inlineTokenDefs = true, using the setProperty() method, during encoding. While decoding, there is no schema required.

The Parts of Binary XML for Java

The Java XML binary functionality has three parts:

- Binary XML encoding - The binary XML encoder converts XML 1.0 infoset to binary XML.
- Binary XML decoding - The binary XML decoder converts binary XML to XML infoset.
- Binary XML vocabulary management, which includes schema management and token management.
Binary XML Encoding

The encoder is created from the BinXMLStream. It takes as input the XML text and outputs the encoded binary XML to the BinXMLStream it was created from. The encoder reads the XML text using streaming SAX. The encoding of the XML text is based on the results of the XML parsing.

Set the schemaAware flag on the encoder that specifies whether the encoding is schema-aware or schema-less.

For schema-aware encoding, the encoder determines whether the schema with the particular schema URL has been registered with the vocabulary manager. For a repository-based or a database-based processor, the encoder queries the repository or the database for the compiled schema based on the schema URL. If the schema is available in the database, it is fetched from the repository or database in the binary XML format and registered with the local vocabulary manager. The vocabulary is schema.

Also set a flag to indicate that the encoding results in a binary XML stream that is independent of a schema. In this case, the resulting binary XML stream contains all token definitions inline and is not dependent on schema or external token sets.

If the encoding is schema-aware, the encoder uses the datatype information from the schema object for more efficient encoding of the SAX stream. There is a default encoding datatype associated with each schema built-in datatype. Binary XML stream encoding using a schema implies at least partial validity with respect to the schema (for partial validity there is no validation for unique key, or keyref, or ID, or DREFs). If the data is known to be completely valid with respect to a schema, the encoded binary XML stream stores this information.

See Also: Oracle XML DB Developer’s Guide for tables of the binary encoding datatypes and their mappings from XML schema datatypes

If there is no schema associated with the text XML, then integer token ids are generated for repeated items in the text XML. Creating a token table of token ids and token definitions is an important compression technique. The token definitions are stored as token tables in the vocabulary cache. If the property for inline token definitions is set, then the token definitions are present inline.

See Also: "Token Management" on page 5-6

Another property on the encoder is specifying PSVI (Post Schema Validated Infoset) information as part of the binary stream. If this is set to true then PSVI information can be accessed using XDK extension APIs for PSVI on DOM. If $psvi = true$ then the input XML is fully validated with respect to the schema. If $psvi$ is $false$ then PSVI information is not included in the output binary stream. The default is $false$.

Binary XML Decoding

The binary XML decoder converts binary XML to XML infoset. The decoder is created from the BinXMLStream; it reads binary XML from this stream and outputs SAX events or provide a pull style InfosetReader API for reading the decoded XML. If a schema is associated with the BinXMLStream, the binary XML decoder retrieves the associated schema object from the vocabulary cache using the vocabulary id before decoding. If the schema is not available in the vocabulary cache, and the connection information to the server is available, then the schema is fetched from the server.
If no schema is associated with BinXMLStream, then the token definitions can be either inline in the BinXMLStream or stored in a token set. If tokens of a corresponding namespace are not stored in the local vocabulary cache, then the token set is fetched from the repository.

**Binary XML Vocabulary Management**

The binary XML processors are of different types depending on where the metadata (schema or token sets) are located - either local binary XML processor or repository binary XML processor.

**Schema Management**

For metadata persistence, it is recommended that you use the DB Binary XML processor. In this case, schemas and token sets are registered with the database. The vocabulary manager fetches the schema or token sets from the database and cache it in the local vocabulary cache for encoding and decoding purposes.

*See Also:* "Binary XML DB" on page 5-8

If you need to use a persistent metadata repository that is not a database, then you can plug in your own metadata repository. You must implement the interface for communicating with this repository, BinXMLMetadataProvider.

**Schema Registration**

Register schemas locally with the local binary XML processor. The local binary XML processor contains a vocabulary manager that maintains all schemas submitted by the user for the duration of its existence. The vocabulary manager associated with a local binary XML processor does not provide for schema persistence.

If you register the same schema (same schema location and same target namespace) then the schema is not parsed, and the existing vocabulary id is returned. If a new schema with the same target namespace and a different schema location is registered then the existing schema definition is augmented with the new schema definitions or results in conflict error.

**Schema Identification**

Each schema is identified by a vocabulary id. The vocabulary id is in the scope of the processor and is unique within the processor. Any document that validates with a schema is required to validate with a latest version of the schema.

**Schema Annotations**

Binary XML annotations can only appear within the `<xsd:appInfo>` element in a schema. There are two categories of schema annotations - User-level and System-level. The vocabulary manager interprets these at the time of schema registration. All other types of annotations (for example, database related annotations, is ignored).

**User-Level Annotations**

These are specified by the user before registration.

encodingType - This can be used within a `xsd:element`, `xsd:attribute` or `xsd:simpleType` elements. It indicates the datatype to be used for encoding the node value of the particular element or attribute. For strings, there is only support for UTF8 encoding in this release.
System-Level Annotations
The vocabulary manager adds these at the time of registration; you cannot overwrite them.

Token Management
Token sets can be fetched from the database or metadata repository, cached in the local vocabulary manager and used for decoding. While encoding, token sets can be pushed to the repository for persistence.

Token definitions can also be included as part of the binary XML stream by setting a flag on the encoder.

Using Java Binary XML Package
A BinXMLStream class represents the binary XML stream. The different storage locations defined for the binary XML Stream are:

- InputStream - stream for reading.
- OutputStream - stream for writing.
- URL - stream for reading.
- File - stream for read and write.
- BLOB - stream for reading and writing.
- Byte array - stream for reading and writing.
- In memory - stream for reading and writing.

The BinXMLStream object specifies the type of storage during creation.

A BinXMLStream object can be created from a BinXMLProcessor factory. This factory can be initialized with a JDBC connection (for remote metadata access), connection pool, URL or a PageManagerPool (for lazy in-memory storage). BinXMLEncoder and BinXMLDecoder can be created from the BinXMLStream for encoding or decoding.

1. Here is an example of creating a processor without a repository, registering a schema, encoding XML SAX events into schema-aware binary format, and storing in a file:

```java
BinXMLProcessor proc = BinXMLProcessorFactory.createProcessor();
proc.registerSchema(schemaURL);
BinXMLStream outbin = proc.createBinaryStream(outFile);
BinXMLEncoder enc = outbin.getEncoder();
enc.setSchemaAware(true);
ContentHandler hdlr = enc.getContentHandler();

In addition to getting the ContentHandler, you can also get the other handlers, such as:

LexicalHandler lexhdlr = enc.getLexicalHandler();
DTDHandler dtdhdlr = encenc.getDTDHandler();
DeclHandler declhdlr = encenc.getDeclHandler();
ErrorHandler errhdlr = encenc.getErrorHandler();

Use hdlr in the application that generates the SAX events.

2. Here is an example of creating a processor with a database repository, decoding a schema-aware binary stream and reading the decoded XML using pull API. The schema is fetched from the database repository for decoding.
Using Java Binary XML Package

Using Binary XML for Java

```java
DBBinXMLMetadataProvider dbrep = 
    BinXMLMetadataProviderFactory.createDBMetadataProvider();
BinXMLProcessor proc = BinXMLProcessorFactory.createProcessor(dbrep);
BinXMLStream inpbin = proc.createBinaryStream(blob);
BinXMLDecoder dec = inpbin.getDecoder();
InfosetReader xmlreader = dec.getReader();
```

Use `xmlreader` to read XML in a pull-style from the decoder.

**Binary XML Encoder**

The encoder takes XML input, which is parsed and read using SAX events, and outputs binary XML.

**Schema-less Option**

You can specify the schema-aware or the schema-less option before encoding. The default is schema-less encoding. If the schema-aware option is set, then the encoding is done based on schema(s) specified in the instance document. The annotated schema(s) used for encoding is also required at the time of decoding. If the schema-less option is specified, then the encoding is independent of schema(s), but the tokens are inline by default. To override the default, set `Inline-token = false`.

**Inline-token Option**

You can set an option to create a binary XML Stream with inline token definitions before encoding. If "inlining" is turned off, than you must ensure that the processors for the encoder or decoder are using the same metadata repository. The flag `Inline-token` is ignored if the schema-aware option is true. By default, the token definitions is inline.

*Figure 5–1 Binary XML Encoding*

**Binary XML Decoder**

The binary XML decoder takes binary XML stream as input and generates SAX Events as output, or provides a pull interface to read the decoded XML. In the case of schema-aware binary XML stream, the binary XML decoder interacts with the vocabulary manager to extract the schema information.

- 
- 

Using Binary XML for Java 5-7
If the vocabulary manager does not contain the required schema, and the processor is of type binary XML DB with a valid JDBC connection, then the remote schema is fetched from the database or the metadata repository based on the vocabulary id in the binary XML stream to be decoded. Similarly, the set of token definitions can be fetched from the database or the metadata repository.

**Figure 5–2 Binary XML Decoder**

![Binary XML Decoder Diagram]

**Schema Registration**

Here is the flow of this process: If the vocabulary is an XML schema; it takes the XML schema text as input. The schema annotator annotates the schema text with system level annotations. The schema might already have some user level annotations.

The resulting annotated schema is processed by the Schema Builder to build an XML schema object. This XML schema object is stored in the vocabulary cache. The vocabulary cache assigns a unique vocabulary id for each XML schema object, which is returned as output. The annotated DOM representation of the schema is sent to the binary XML encoder.

**Resolving xsi:schemaLocation**

During encoding, if `schemaAware` is true and the property `ImplicitSchemaRegistration` is true, then the first `xsi:schemaLocation` tag present in the root element of an XML instance document automatically registers that schema in the local vocabulary manager. All other `schemaLocation` tags are not explicitly registered. If the processor is database-oriented, then the schema is also registered in the database; similarly for any metadata repository based processor.

If the encoding is set to `schemaAware` is false or `ImplicitSchemaRegistration` is false, then all `xsi:schemaLocation` tags are ignored by the encoder.

**Binary XML DB**

A `DBBinXMLMetadataProvider` object is either instantiated with a dedicated JDBC connection or a connection pool to access vocabulary information such as schema and token set. The processor is also associated with one or more data connections to access XML data.

A binary XML Processor can communicate with the database for various types of binary XML operations involving storage and retrieval of binary XML schemas, token...
sets, and binary XML streams. Database communication is involved in the following ways:

1. Extracting compiled binary XML Schema using the vocabulary id or the schema URL

   To retrieve a compiled binary XML schema for encoding, the database is queried based on the schema URL. For decoding the binary XML schema, fetch it from the database based on the vocabulary id.

2. Storing noncompiled binary XML schema using the schema URL and retrieving the vocabulary id.

   When the xsi:schemaLocation tag is encountered during encoding, the schema is registered in the database for persistent storage in the database. The vocabulary id associated with the schema, as well as the binary version of the compiled schema is retrieved back from the database; the compiled schema object is built and stored in the local cache using the vocabulary id returned from the database.

3. Retrieving a binary token set using namespace URL.

   If a binary stream to be decoded is associated with token tables for decoding, these are fetched from the database using the metadata connection.

4. Storing binary token set using namespace URL

   If the XML text has been encoded without a schema, then it results in a token set of token definitions. These token tables can be stored persistently in the database. The metadata connection is used for transferring the token set to the database.

5. Binary XML stream with remote storage option

   It is your responsibility to create a table containing an XMLType column with binary XML for storing the result of encoding and retrieving the binary XML for decoding. Communication with the database can be achieved with SQL*Net and JDBC. Fetch the XMLType object from the output result set of the JDBC query. The BinXMLStream for reading the binary data or for writing out binary data can be created from the XMLType object. The XMLType class needs to be extended to support reading and writing of binary XML data.

**Persistent Storage of Metadata**

A local vocabulary manager and cache stores metadata information in the memory for the life of the BinXMLProcessor. Plug in your own back-end storage for metadata by implementing the BinXMLMetadataProvider interface and plugging it into the BinXMLProcessor. Currently only one metadata provider for each processor is supported.

You must code a FileBinXMLMetadataProvider that implements the BinXMLMetadataProvider interface. The encoder and decoder uses these APIs to access metadata from the persisted back-end storage. Set up the configuration information for the persistent storage: for example, root directory in the case of a file system in FileBinXMLMetadataProvider class. Instantiate FileBinXMLMetadataProvider and plug it into the BinXMLProcessor.
This chapter contains these topics:

- Introduction to the XSLT Processor
- Using the XSLT Processor for Java: Overview
- Transforming XML
- Programming with Oracle XSLT Extensions
- Tips and Techniques for Transforming XML

**Introduction to the XSLT Processor**

This section contains the following topics:

- Prerequisites
- Standards and Specifications
- XML Transformation with XSLT 1.0 and 2.0

**Prerequisites**

XSLT is an XML-based language that you can use to transform one XML document into another text document. For example, you can use XSLT to accept an XML data document as input, perform arithmetic calculations on element values in the document, and generate an XHTML document that shows the calculation results.

In XSLT, XPath is used to navigate and process elements in the source node tree. XPath models an XML document as a tree made up of nodes; the types of nodes in the XPath node tree correspond to the types of nodes in a DOM tree.

This chapter assumes that you are familiar with the following W3C standards:

- eXtensible Stylesheet Language (XSL) and eXtensible Stylesheet Language Transformation (XSLT). If you require a general introduction to XSLT, consult the XML resources listed in “Related Documents” on page xxxvii of the preface.

**Standards and Specifications**

XSLT is currently available in two versions: a working draft for XSLT 2.0 and the XSLT 1.0 Recommendation. You can find the specifications at the following URLs:

- [http://www.w3.org/TR/xslt20/](http://www.w3.org/TR/xslt20/)
XPath, which is the navigational language used by XSLT and other XML languages, is available in two versions: a working draft for XPath 2.0 and the XPath 1.0 Recommendation. You can find the specifications for the two XPath versions at the following URLs:

- [http://www.w3.org/TR/xpath20/](http://www.w3.org/TR/xpath20/)
- [http://www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)

Oracle XDK XSLT processor implements both the XSLT and XPath 1.0 standards as well as the current working drafts of the XSLT and XPath 2.0 standards. The XDK XSLT processor supports the XPath 2.0 functions and operators. You can find the specification at the following URL:

[http://www.w3.org/TR/xpath-functions/](http://www.w3.org/TR/xpath-functions/)

**See Also:** Chapter 31, "XDK Standards" for a summary of the standards supported by the XDK

**XML Transformation with XSLT 1.0 and 2.0**

In Oracle Database 10g, the XDK provides several useful features not included in XSLT 1.0. To use XSLT 2.0, set the `version` attribute in your stylesheet as follows:

```xml
<?xml-stylesheet version='2.0' ... ?>
```

Some of the most useful XSLT 2.0 features are the following:

- **User-defined functions**
  
  You can use the `<xsl:function>` declaration to define functions. This element must have one `name` attribute to define the function name. The value of the `name` attribute is a QName. The content of the `<xsl:function>` element is zero or more `<xsl:param>` elements that specify the formal arguments of the function, followed by a sequence constructor that defines the value returned by the function.

  Note that QName can have a null namespace, but user-defined functions must have a non-null namespace. That is, if abc is defined as a namespace, then add is not a legal user-defined function, but abc: add is.

- **Grouping**
  
  You can use the `<xsl:for-each-group>` element, `current-group()` function, and `current-grouping-key()` function to group items.

- **Multiple result documents**
  
  You can use the `<xsl:result-document>` element to create a result tree. The content of the `<xsl:result-document>` element is a sequence constructor for the children of the document node of the tree.

  For example, this element enables you to accept an XML document as input and break it into separate documents. You can take an XML document that describes a list of books and generate an XHTML document for each book. You can then validate each output document.

- **Temporary trees**
  
  Instead of representing the intermediate XSL transformation results and XSL variables as strings, as in XSLT 1.0, you can store them as a set of document nodes. The document nodes, which you can construct with the `<xsl:variable>`, `<xsl:param>`, and `<xsl:with-param>` elements, are called temporary trees.
Character mapping

In XSLT 1.0, you had to use the `disable-output-escaping` attribute of the `<xsl:text>` and `<xsl:value-of>` elements to specify character escaping. In XSLT 2.0, you can declare mapping characters with an `<xsl:character-map>` element as a top level stylesheet element. You can use this element to generate files with reserved or invalid XML characters in the XSLT outputs, such as `<`, `>`, and `&`.

See Also: [http://www.w3.org/TR/xslt20](http://www.w3.org/TR/xslt20) for explanation and examples of XSLT 2.0 features

Using the XSLT Processor for Java: Overview

The Oracle XDK XSLT processor is a software program that transforms an XML document into another text-based format. For example, the processor can transform XML into XML, HTML, XHTML, or text. You can invoke the processor programmatically by using the APIs or run it from the command line. The XSLT processor can perform the following tasks:

- Reads one or more XSLT stylesheets. The processor can apply multiple stylesheets to a single XML input document and generate different results.
- Reads one or more input XML documents. The processor can use a single stylesheet to transform multiple XML input documents.
- Builds output documents by applying the rules in the stylesheet to the input XML documents. The output is a DOM tree, output stream, or series of SAX events.

Whereas XSLT is a function-based language that generally requires a DOM of the input document and stylesheet to perform the transformation, the Java XDK implementation of the XSLT processor can use SAX to create a stylesheet object to perform transformations with higher efficiency and fewer resources. You can reuse this stylesheet object to transform multiple documents without reparsing the stylesheet.

Using the XSLT Processor: Basic Process

Figure 6–1 depicts the basic design of the XSLT processor for Java.

See Also: [Oracle Database XML Java API Reference](http://www.oracle.com) to learn about the XMLParser and XSDBuilder classes
Running the XSLT Processor Demo Programs

Demo programs for the XSLT processor for Java are included in `$ORACLE_HOME/xdk/demo/java/parser/xslt`. Table 6–1 describes the XML files and programs that you can use to test the XSLT processor.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>match.xml</td>
<td>A sample XML document that you can use to test ID selection and pattern matching. Its associated stylesheet is <code>match.xsl</code>.</td>
</tr>
<tr>
<td>match.xsl</td>
<td>A sample stylesheet for use with <code>match.xml</code>. You can use it to test simple identity transformations.</td>
</tr>
<tr>
<td>math.xml</td>
<td>A sample XML data document that you can use to perform simple arithmetic. Its associated stylesheet is <code>math.xsl</code>.</td>
</tr>
<tr>
<td>math.xsl</td>
<td>A sample stylesheet for use with <code>math.xml</code>. The stylesheet outputs an HTML page with the results of arithmetic operations performed on element values in <code>math.xml</code>.</td>
</tr>
<tr>
<td>number.xml</td>
<td>A sample XML data document that you can use to test for source tree numbering. The document describes the structure of a book.</td>
</tr>
<tr>
<td>number.xsl</td>
<td>A sample stylesheet for use with <code>number.xml</code>. The stylesheet outputs an HTML page that calculates section numbers for the sections in the book described by <code>number.xml</code>.</td>
</tr>
<tr>
<td>position.xml</td>
<td>A sample XML data document that you can use to test for <code>position()=X</code> in complex patterns. Its associated stylesheet is <code>position.xsl</code>.</td>
</tr>
</tbody>
</table>
Documentation for how to compile and run the sample programs is located in the README. The basic steps are as follows:

1. Change into the $ORACLE_HOME/xdk/demo/java/parser/xslt directory (UNIX) or %ORACLE_HOME%\xdk\demo\java\parser\xslt directory (Windows).

2. Make sure that your environment variables are set as described in "Setting Up the Java XDK Environment" on page 3-5.

3. Run make (UNIX) or Make.bat (Windows) at the command line. The make file compiles the source code and then runs the XSLSample and XSLSample2 programs for each *.xml file and its associated *.xsl stylesheet. The program writes its output for each transformation to *.out.

4. You can view the *.out files to see the output for the XML transformations. You can also run the programs on the command line as follows, where name is replaced by match, math, and so forth:

   java XSLSample name.xsl name.xml
   java XSLSample2 name.xsl name.xml

   For example, run the match.xml demos as follows:

   java XSLSample match.xsl match.xml
   java XSLSample2 match.xsl match.xml

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>position.xsl</td>
<td>A sample stylesheet for use with position.xml. The stylesheet outputs an HTML page with the results of complex pattern matching.</td>
</tr>
<tr>
<td>reverse.xml</td>
<td>A sample XML data document that you can use with reverse.xsl to traverse backward through a tree.</td>
</tr>
<tr>
<td>reverse.xsl</td>
<td>A sample stylesheet for use with reverse.xml. The stylesheet output the item numbers in reverse.xml in reverse order.</td>
</tr>
<tr>
<td>string.xml</td>
<td>A sample XML data document that you can use to test perform various string test and manipulations. Its associated stylesheet is string.xsl.</td>
</tr>
<tr>
<td>string.xsl</td>
<td>A sample stylesheet for use with string.xml. The stylesheet outputs an XML document that displays the results of the string manipulations.</td>
</tr>
<tr>
<td>style.txt</td>
<td>A stylesheet that provides the framework for an HTML page. The stylesheet is included by number.xsl.</td>
</tr>
<tr>
<td>variable.xml</td>
<td>A sample XML data document that you can use to test the use of XSL variables. The document describes the structure of a book. Its associated stylesheet is variable.xsl.</td>
</tr>
<tr>
<td>variable.xsl</td>
<td>A stylesheet for use with variable.xml. The stylesheet makes extensive use of XSL variables.</td>
</tr>
<tr>
<td>XSLSample.java</td>
<td>A sample application that offers a simple example of how to use the XSL processing capabilities of the Oracle XSLT processor. The program transforms an input XML document by using an input stylesheet. This program builds the result of XSL transformations as a DocumentFragment and does not show xsl:output features. Run this program with any XSLT stylesheet in the directory as a first argument and its associated *.xml XML document as a second argument. For example, run the program with variable.xsl and variable.xml or string.xsl and string.xml.</td>
</tr>
<tr>
<td>XSLSample2.java</td>
<td>A sample application that offers a simple example of how to use the XSL processing capabilities of the Oracle XSLT processor. The program transforms an input XML document by using an input stylesheet. This program outputs the result to a stream and supports xsl:output features. Like XSLSample.java, you can run it against any pair of XML data documents and styleheets in the directory.</td>
</tr>
</tbody>
</table>
Using the XSLT Processor Command-Line Utility

The XDK includes oraxsl, which is a command-line Java interface that can apply a stylesheet to multiple XML documents. The $ORACLE_HOME/bin/oraxsl and %ORACLE_HOME%\bin\oraxsl.bat shell scripts execute the oracle.xml.jaxb.oraxsl class. To use oraxsl ensure that your CLASSPATH is set as described in "Setting Up the Java XDK Environment" on page 3-5.

Use the following syntax on the command line to invoke oraxsl:

```
oraxsl options source stylesheet result
```

The oraxsl utility expects a stylesheet, an XML file to transform, and an optional result file. If you do not specify a result file, then the utility sends the transformed document to standard output. If multiple XML documents need to be transformed by a stylesheet, then use the -l or -d options in conjunction with the -s and -r options. These and other options are described in Table 6–2.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-w</td>
<td>Shows warnings. By default, warnings are turned off.</td>
</tr>
<tr>
<td>-e error_log</td>
<td>Specifies file into which the program writes errors and warnings.</td>
</tr>
<tr>
<td>-l xml_file_list</td>
<td>Lists files to be processed.</td>
</tr>
<tr>
<td>-d directory</td>
<td>Specifies the directory that contains the files to transform. The default behavior is to process all files in the directory. If only a subset of the files in that directory, for example, one file, need to be processed, then change this behavior by setting -l and specifying the files that need to be processed. You can also change the behavior by using the -x or -i option to select files based on their extension.</td>
</tr>
<tr>
<td>-x source_extension</td>
<td>Specifies extensions for the files that should be excluded. Use this option in conjunction with -d. The program does not select any files with the specified extension.</td>
</tr>
<tr>
<td>-i source_extension</td>
<td>Specifies extensions for the files that should be included. Use this option in conjunction with -d. The program selects only files with the specified extension.</td>
</tr>
<tr>
<td>-s stylesheet</td>
<td>Specifies the stylesheet. If you set -d or -l, then set -s to indicate the stylesheet to be used. You must specify the complete path.</td>
</tr>
<tr>
<td>-r result_extension</td>
<td>Specifies the extension to use for results. If you set -d or -l, then set -r to specify the extension to be used for the results of the transformation. So, if you specify the extension out, the program transformed an input document doc to doc.out. By default, the program places the results in the current directory. You can change this behavior by using the -o option, which allows you to specify a directory for the results.</td>
</tr>
<tr>
<td>-o result_directory</td>
<td>Specifies the directory in which to place results. You must set this option in conjunction with the -r option.</td>
</tr>
<tr>
<td>-p param_list</td>
<td>Lists parameters.</td>
</tr>
<tr>
<td>-t num_of_threads</td>
<td>Specifies the number of threads to use for processing. Using multiple threads can provide performance improvements when processing multiple documents.</td>
</tr>
<tr>
<td>-v</td>
<td>Generates verbose output. The program prints some debugging information and can help in tracing any problems that are encountered during processing.</td>
</tr>
<tr>
<td>-debug</td>
<td>Generates debugging output. By default, debug mode is disabled. Note that a GUI version of the XSLT debugger is available in Oracle JDeveloper.</td>
</tr>
</tbody>
</table>

Using the XSLT Processor Command-Line Utility: Example

You can test oraxsl on the various XML files and stylesheets in $ORACLE_HOME/xdk/demo/java/parser/xslt. Example 6–1 displays the contents of math.xml.
Example 6–1  math.xml

```xml
<?xml version="1.0"?>
<doc>
  <n1>5</n1>
  <n2>2</n2>
  <div>-5</div>
  <mod>2</mod>
</doc>
```

The XSLT stylesheet named math.xsl is shown in Example 6–2.

Example 6–2  math.xsl

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="doc">
    <HTML>
      <H1>Test for mod.</H1>
      <HR/>
      <P>Should say "1": <xsl:value-of select="5 mod 2"/></P>
      <P>Should say "1": <xsl:value-of select="n1 mod n2"/></P>
      <P>Should say "-1": <xsl:value-of select="div mod mod"/></P>
      <P><xsl:value-of select="div or ((mod)) | or"/></P>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```

You can run the oraxsl utility on these files to produce HTML output as shown in the following example:

oraxsl math.xml math.xsl math.htm

The output file math.htm is shown in Example 6–3.

Example 6–3  math.htm

```html
<HTML>
  <H1>Test for mod.</H1>
  <HR/>
  <P>Should say "1": 1</P>
  <P>Should say "1": 1</P>
  <P>Should say "-1": -1</P>
  <P>true</P>
</HTML>
```

Transforming XML

This section contains the following topics:

- Performing Basic XSL Transformation
- Obtaining DOM Results from an XSL Transformation

Performing Basic XSL Transformation

As explained in "Using the XSLT Processor for Java: Overview" on page 6-3, the fundamental classes used by the XSLT processor are DOMParser and XSLProcessor. The XSL2Sample.java demo program provides a good illustration of how to use these classes to transform an XML document with an XSLT stylesheet.
Use the following basic steps to write Java programs that use the XSLT processor:

1. Create a DOM parser object that you can use to parse the XML data documents and XSLT stylesheets. The following code fragment from XSL2Sample.java illustrates how to instantiate a parser:

   ```java
   XMLDocument xml, xsldoc, out;
   URL xslURL;
   URL xmlURL;
   // ...
   parser = new DOMParser();
   parser.setPreserveWhitespace(true);
   
   Note that by default, the parser does not preserve whitespace unless a DTD is used. It is important to preserve whitespace because it enables XSLT whitespace rules to determine how whitespace is handled.
   
2. Parse the XSLT stylesheet with the DOMParser.parse() method. The following code fragment from XSL2Sample.java illustrates how to perform the parse:

   ```java
   xslURL = DemoUtil.createURL(args[0]);
   parser.parse(xslURL);
   xsldoc = parser.getDocument();
   
   3. Parse the XML data document with the DOMParser.parse() method. The following code fragment from XSL2Sample.java illustrates how to perform the parse:

   ```java
   xmlURL = DemoUtil.createURL(args[1]);
   parser.parse(xmlURL);
   xml = parser.getDocument();
   
   4. Create a new XSLT stylesheet object. You can pass objects of the following classes to the XSLProcessor.newXSLStylesheet() method:

   - java.io.Reader
   - java.io.InputStream
   - XMLDocument
   - java.net.URL

   For example, XSL2Sample.java illustrates how to create a stylesheet object from an XMLDocument object:

   ```java
   XSLProcessor processor = new XSLProcessor();
   processor.setBaseURL(xslURL);
   XSLStylesheet xsl = processor newXSLStylesheet(xsldoc);
   
   5. Set the XSLT processor to display any warnings. For example, XSL2Sample.java calls the showWarnings() and setErrorStream() methods as follows:

   ```java
   processor.showWarnings(true);
   processor.setErrorStream(System.err);
   
   6. Use the XSLProcessor.processXSL() method to apply the stylesheet to the input XML data document. Table 6–3 lists some of the other available XSLProcessor methods.
The following code fragment from XSL2Sample.java shows how to apply the stylesheet to the XML document:

```java
processor.processXSL(xsl, xml, System.out);
```

7. Process the transformed output. You can transform the results by creating an XML document object, writing to an output stream, or reporting SAX events.

The following code fragment from XSL2Sample.java shows how to print the results:

```java
processor.processXSL(xsl, xml, System.out);
```

See Also:

- [http://www.w3.org/TR/xslt](http://www.w3.org/TR/xslt)
- [http://www.w3.org/style/XSL](http://www.w3.org/style/XSL)
- Chapter 4, "XML Parsing for Java"

## Obtaining DOM Results from an XSL Transformation

The XSLSample.java demo program illustrates how to generate an oracle.xml.parser.v2.XMLDocumentFragment object as the result of an XSL transformation. An XMLDocumentFragment is a "lightweight" Document object that extracts a portion of an XML document tree. The XMLDocumentFragment class implements the org.w3c.dom.DocumentFragment interface.

The XSL2Sample.java program illustrates how to generate a DocumentFragment object. The basic steps for transforming XML are the same as those described in "Performing Basic XSL Transformation" on page 6-7. The only difference is in the arguments passed to the XSLProcessor.processXSL() method. The following code fragment from XSL2Sample.java shows how to create the DOM fragment and then print it to standard output:

```java
XMLDocumentFragment result = processor.processXSL(xsl, xml);
result.print(System.out);
```

Table 6–4 lists some of the XMLDocumentFragment methods that you can use to manipulate the object.

<table>
<thead>
<tr>
<th><strong>Table 6–4 XMLDocumentFragment Methods</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Method</strong></td>
</tr>
<tr>
<td>getAttributes()</td>
</tr>
</tbody>
</table>

The following code fragment from XSL2Sample.java shows how to apply the stylesheet to the XML document:

```java
processor.processXSL(xsl, xml, System.out);
```
Overview of Oracle XSLT Extensions

The XSLT 1.0 standard defines two kinds of extensions: extension elements and extension functions. The XDK provides extension functions for XSLT processing that enable users of the XSLT processor to call any Java method from XSL expressions. Note the following guidelines when using Oracle XSLT extensions:

- When you define an XSLT extension in a given programming language, you can only use the XSLT stylesheet with XSLT processors that can invoke this extension. Thus, only the Java version of the processor can invoke extension functions that are defined in Java.
- Use XSLT extensions only if the built-in XSL functions cannot solve a given problem.
- As explained in the following section, the namespace of the extension class must start with the proper URL.

The following Oracle extension functions are particularly useful:

- `<ora:output>`, you can use `<ora:output>` as a top-level element or in an XSL template. If used as a top-level element, it is similar to the `<xsl:output>` extension function, except that it has an additional `name` attribute. When used as a template, it has the additional attributes `use` and `href`. This function is useful for creating multiple outputs from one XSL transformation.
- `<ora:node-set>`, which converts a result tree fragment into a node-set. This function is useful when you want to refer the existing text or intermediate text results in XSL for further transformation.
Specifying Namespaces for XSLT Extension Functions

The Oracle Java extension functions belong to the namespace that corresponds to the following URI:

http://www.oracle.com/XSL/Transform/java/

An extension function that belongs to the following namespace refers to methods in the Java classname, so that you can construct URIs in the following format:

http://www.oracle.com/XSL/Transform/java/classname

For example, you can use the following namespace to call java.lang.String methods from XSL expressions:

http://www.oracle.com/XSL/Transform/java/java.lang.String

---

**Note:** When assigning the xsl prefix to a namespace, the correct URI is xmlns:xsl="http://www.w3.org/1999/XSL/Transform". Any other URI fails to give correct output.

---

Using Static and Non-Static Java Methods in XSLT

If the Java method is a non-static method of the class, then the first parameter is used as the instance on which the method is invoked, and the rest of the parameters are passed to the method. If the extension function is a static method, however, then all the parameters of the extension function are passed as parameters to the static function. Example 6–4 shows how to use the java.lang.Math.ceil() method in an XSLT stylesheet.

**Example 6–4 Using a Static Function in an XSLT Stylesheet**

```xml
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   <xsl:template match="/">
     <xsl:value-of select="math:ceil('12.34')"/>
   </xsl:template>
</xsl:stylesheet>
```

For example, you can create Example 6–4 as stylesheet ceil.xsl and then apply it to any well-formed XML document. For example, run the oraxsl utility as follows:

    oraxsl ceil.xsl ceil.xsl ceil.out

The output document ceil.out has the following content:

```xml
<?xml version = '1.0' encoding = 'UTF-8'?>
13
```

---

**Note:** The XSL class loader only knows about statically added JARs and paths in the CLASSPATH as well as those specified by wrapper.classpath. Files added dynamically are not visible to XSLT processor.
Using Constructor Extension Functions

The extension function `new` creates a new instance of the class and acts as the constructor. Example 6–5 creates a new `String` object with the value ”Hello World,” stores it in the XSL variable `str1`, and then outputs it in uppercase.

Example 6–5  Using a Constructor in an XSLT Stylesheet

```xml
<xsl:stylesheet version='1.0' 
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:jstring="http://www.oracle.com/XSL/Transform/java/java.lang.String">
  <xsl:template match="/">
    <!-- creates a new java.lang.String and stores it in the variable str1 -->
    <xsl:variable name="str1" select="jstring:new('HeLlO wOrLd')"/>
    <xsl:value-of select="jstring:toUpperCase($str1)"/>
  </xsl:template>
</xsl:stylesheet>
```

For example, you can create this stylesheet as `hello.xsl` and apply it to any well-formed XML document. For example, run the `oraxsl` utility as follows:

```
oraxsl hello.xsl hello.xsl hello.out
```

The output document `hello.out` has the following content:

```xml
<?xml version = '1.0' encoding = 'UTF-8'?>
HELLO WORLD
```

Using Return Value Extension Functions

The result of an extension function can be of any type, including the five types defined in XSL and the additional simple XML Schema data types defined in XSLT 2.0:

- `NodeSet`
- `Boolean`
- `String`
- `Number`
- `ResultTree`

You can store these data types in variables or pass to other extension functions. If the result is of one of the five types defined in XSL, then the result can be returned as the result of an XSL expression.

The XSLT Processor supports overloading based on the number of parameters and type. The processor performs implicit type conversion between the five XSL types as defined in XSL. It performs type conversion implicitly among the following datatypes, and also from `NodeSet` to the following datatypes:

- `String`
- `Number`
- `Boolean`
- `ResultTree`

Overloading based on two types that can be implicitly converted to each other is not permitted. The following overloading results in an error in XSL because `String` and `Number` can be implicitly converted to each other:

```java
 overloadme(int i){}
```
Mapping between XSL datatypes and Java datatypes is done as follows:

- **String** -> `java.lang.String`
- **Number** -> `int, float, double`
- **Boolean** -> `boolean`
- **NodeSet** -> `NodeList`
- **ResultTree** -> `XMLDocumentFragment`

The stylesheet in Example 6–6 parses the `variable.xml` document, which is located in the directory `$ORACLE_HOME/xdk/demo/java/parser/xslt`, and retrieves the value of the `<title>` child of the `<chapter>` element.

**Example 6–6  gettitle.xsl**

```xml
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:parser = "http://www.oracle.com/XSL/Transform/java/oracle.xml.parser.v2.DOMParser"
    <xsl:template match="/">
        <!-- Create a new instance of the parser and store it in myparser variable -->
        <xsl:variable name="myparser" select="parser:new()"/>
        <!-- Call an instance method of DOMParser. The first parameter is the object. The PI is equivalent to $myparser.parse('file:/my_path/variable.xml'). Note that you should replace my_path with the absolute path on your system. -->
        <xsl:value-of select="parser:parse($myparser, 'file:/my_path/variable.xml')"/>
        <!-- Get the document node of the XML Dom tree -->
        <xsl:variable name="mydocument" select="parser:getDocument($myparser)"/>
        <!-- Invoke getelementsbytagname on mydocument -->
        <xsl:for-each select="document:getElementsByTagName($mydocument,'chapter')">
            The value of the title element is: <xsl:value-of select="docinfo/title" />
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

You can create Example 6–6 as `gettitle.xsl` and then run `oraxsl` as follows:

```
oraxsl gettitle.xsl gettitle.xsl variable.out
```

The output document `variable.out` has the following content:

```xml
<?xml version = '1.0' encoding = 'UTF-8'>
The value of the title element is: Section Tests
```

**Tips and Techniques for Transforming XML**

This section lists XSL and XSLT Processor for Java hints, and contains these topics:

- Merging XML Documents with XSLT
- Creating an HTML Input Form Based on the Columns in a Table
Merging XML Documents with XSLT

"Merging Documents with appendChild()" on page 4–46 discusses the DOM technique for merging documents. If the merging operation is simple, then you can also use an XSLT-based approach. Suppose that you want to merge the XML documents in Example 6–7 and Example 6–8.

Example 6–7  msg_w_num.xml

```xml
<messages>
  <msg>
    <key>AAA</key>
    <num>01001</num>
  </msg>
  <msg>
    <key>BBB</key>
    <num>01011</num>
  </msg>
</messages>
```

Example 6–8  msg_w_text.xml

```xml
<messages>
  <msg>
    <key>AAA</key>
    <text>This is a Message</text>
  </msg>
  <msg>
    <key>BBB</key>
    <text>This is another Message</text>
  </msg>
</messages>
```

Example 6–9 displays a sample stylesheet that merges the two XML documents based on matching the <key/> element values.

Example 6–9  msgmerge.xsl

```xml
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output indent="yes"/>
  <!-- store msg_w_text.xml in doc2 variable -->
  <xsl:variable name="doc2" select="document('msg_w_text.xml')"/>

  <!-- match each node in input xml document, that is, msg_w_num.xml -->
  <xsl:template match="@*|node()">
    <!-- copy the current node to the result tree -->
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>

  <!-- match each <msg> element in msg_w_num.xml -->
  <xsl:template match="msg">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
      <xsl:text>  </xsl:text>
      <text><xsl:value-of select="$doc2/messages/msg[.]/text"/>
    </text>
  </xsl:copy>
</xsl:stylesheet>
```
Create the XML files in Example 6–7, Example 6–8, and Example 6–9 and run the following at the command line:

```
oraxsl msg_w_num.xml msgmerge.xsl msgmerge.xml
```

Example 6–10 shows the output document, which merges the data contained in `msg_w_num.xml` and `msg_w_text.xml`.

### Example 6–10 msgmerge.xml

```xml
<?xml version = '1.0' encoding = 'UTF-8'?>
<messages>
  <msg>
    <key>AAA</key>
    <num>01001</num>
    <text>This is a Message</text>
  </msg>
  <msg>
    <key>BBB</key>
    <num>01011</num>
    <text>This is another Message</text>
  </msg>
</messages>
```

This technique is not as efficient for larger files as an equivalent database join of two tables, but it is useful if you have only XML files to work with.

### Creating an HTML Input Form Based on the Columns in a Table

Suppose that you want to generate an HTML form for inputting data that uses column names from a database table. You can achieve this goal by using XSU to obtain an XML document based on the `user_tab_columns` table and XSLT to transform the XML into an HTML form.

1. **Use XSU to generate an XML document based on the columns in the table.** For example, suppose that the table is `hr.employees`. You can run XSU from the command line as follows:

   ```
   java OracleXML getXML -user "hr/password"
   "SELECT column_name FROM user_tab_columns WHERE table_name = 'EMPLOYEES'
   ```

2. **Save the XSU output as an XML file called `emp_columns.xml`.** The XML should look like the following, with one `<ROW>` element corresponding to each column in the table (some `<ROW>` elements have been removed to conserve space):

   ```xml
   <?xml version = '1.0'?>
   <ROWSET>
     <ROW num="1">
       <COLUMN_NAME>EMPLOYEE_ID</COLUMN_NAME>
     </ROW>
     <ROW num="2">
       <COLUMN_NAME>FIRST_NAME</COLUMN_NAME>
     </ROW>
     <!-- rows 3 through 10 -->
     <ROW num="11">
       <COLUMN_NAME>DEPARTMENT_ID</COLUMN_NAME>
     </ROW>
   </ROWSET>
   ```
3. Create a stylesheet to transform the XML into HTML. For example, create the columns.xsl stylesheet as follows:

```xml
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <HTML>
      <xsl:apply-templates select="@*|node()"/>
    </HTML>
  </xsl:template>
  <xsl:template match="ROW">
    <xsl:value-of select="COLUMN_NAME"/>
    <xsl:text>&nbsp;</xsl:text>
    <INPUT NAME="{COLUMN_NAME}"/>
    <BR/>
  </xsl:template>
</xsl:stylesheet>
```

4. Run the oraxsl utility to generate the HTML form. For example:

```
oraxsl emp_columns.xml columns.xsl emp_form.htm
```

5. Review the output HTML form, which should have the following contents:

```html
<HTML>
  EMPLOYEE_ID&nbsp;<INPUT NAME="EMPLOYEE_ID"><BR>
  FIRST_NAME&nbsp;<INPUT NAME="FIRST_NAME"><BR>
  LAST_NAME&nbsp;<INPUT NAME="LAST_NAME"><BR>
  EMAIL&nbsp;<INPUT NAME="EMAIL"><BR>
  PHONE_NUMBER&nbsp;<INPUT NAME="PHONE_NUMBER"><BR>
  HIRE_DATE&nbsp;<INPUT NAME="HIRE_DATE"><BR>
  JOB_ID&nbsp;<INPUT NAME="JOB_ID"><BR>
  SALARY&nbsp;<INPUT NAME="SALARY"><BR>
  COMMISSION_PCT&nbsp;<INPUT NAME="COMMISSION_PCT"><BR>
  MANAGER_ID&nbsp;<INPUT NAME="MANAGER_ID"><BR>
  DEPARTMENT_ID&nbsp;<INPUT NAME="DEPARTMENT_ID"><BR>
</HTML>
```
Using the Schema Processor for Java

This chapter contains these topics:

■ Introduction to XML Validation
■ Using the XML Schema Processor: Overview
■ Validating XML with XML Schemas
■ Tips and Techniques for Programming with XML Schemas

Introduction to XML Validation

This section describes the different techniques for XML validation. It discusses the following topics:

■ Prerequisites
■ Standards and Specifications
■ XML Validation with DTDs
■ XML Validation with XML Schemas
■ Differences Between XML Schemas and DTDs

Prerequisites

This chapter assumes that you have working knowledge of the following technologies:

■ Document Type Definition (DTD). An XML DTD defines the legal structure of an XML document.

If you are unfamiliar with these technologies or need to refresh your knowledge, you can consult the XML resources in "Related Documents" on page xxxvii of the preface.

See Also:

■ http://www.w3schools.com/dtd/ for a DTD tutorial
■ http://www.w3schools.com/schema for an XML Schema language tutorial

Standards and Specifications

XML Schema is a W3C standard. You can find the XML Schema specifications at the following locations:
The Oracle XML Schema processor supports the W3C XML Schema specifications.

See Also: Chapter 31, "XDK Standards" for a summary of the standards supported by the XDK

XML Validation with DTDs

DTDs were originally developed for SGML. XML DTDs are a subset of those available in SGML and provide a mechanism for declaring constraints on XML markup. XML DTDs enable the specification of the following:

- Which elements can be in your XML documents
- The content model of an XML element, that is, whether the element contains only data or has a set of subelements that defines its structure. DTDs can define whether a subelement is optional or mandatory and whether it can occur only once or multiple times.
- Attributes of XML elements. DTDs can also specify whether attributes are optional or mandatory.
- Entities that are legal in your XML documents.

An XML DTD is not itself written in XML, but is a context-independent grammar for defining the structure of an XML document. You can declare a DTD in an XML document itself or in a separate file from the XML document.

Validation is the process by which you verify an XML document against its associated DTD, ensuring that the structure, use of elements, and use of attributes are consistent with the definitions in the DTD. Thus, applications that handle XML documents can assume that the data matches the definition.

By using the XDK, you can write an application that includes a validating XML parser, that is, a program that parses and validates XML documents against a DTD. Note the following aspects of parsers that perform DTD validation:

- Depending on its implementation, a validating parser may stop processing when it encounters an error, or continue.
- A validating parser may report warnings and errors as they occur as in summary form at the end of processing.
- Most processors can enable or disable validation mode, but they must still process entity definitions and other constructs of DTDs.

DTD Samples in the XDK

Example 7–1 shows the contents of a DTD named family.dtd, which is located in $ORACLE_HOME/xdk/demo/java/parser/common/. The <ELEMENT> tags specify the legal nomenclature and structure of elements in the document, whereas the <ATTLIST> tags specify the legal attributes of elements.

Example 7–1   family.dtd

```
<?xml version="1.0" encoding='UTF-8'?>
```
Example 7–2 shows the contents of an XML document named family.xml, which is also located in $ORACLE_HOME/xdk/demo/java/parser/common/. The <!DOCTYPE> element in family.xml specifies that this XML document conforms to the external DTD named family.dtd.

Example 7–2 family.xml

```xml
<?xml version="1.0" standalone="no"?>
<!DOCTYPE family SYSTEM "family.dtd">
<family lastname="Smith">
  <member memberid="m1">Sarah</member>
  <member memberid="m2">Bob</member>
  <member memberid="m3" mom="m1" dad="m2">Joanne</member>
  <member memberid="m4" mom="m1" dad="m2">Jim</member>
</family>
```

XML Validation with XML Schemas

The XML Schema language, also known as XML Schema Definition, was created by the W3C to use XML syntax to describe the content and the structure of XML documents. An XML schema is an XML document written in the XML Schema language. An XML schema document contains rules describing the structure of an input XML document, called an instance document. An instance document is valid if and only if it conforms to the rules of the XML schema.

The XML Schema language defines such things as the following:

- Which elements and attributes are legal in the instance document
- Which elements can be children of other elements
- The order and number of child elements
- Datatypes for elements and attributes
- Default and fixed values for elements and attributes

A validating XML parser tries to determine whether an instance document conforms to the rules of its associated XML schema. By using the XDK, you can write a validating parser that performs this schema validation. Note the following aspects of parsers that perform schema validation:

- Depending on its implementation, the parser may stop processing when it encounters an error, or continue.
- The parser may report warnings and errors as they occur as in summary form at the end of processing.
- The processor must take into account entity definitions and other constructs that are defined in a DTD that is included by the instance document. The XML Schema language does not define what must occurs when an instance document includes both an XML schema and a DTD. Thus, the behavior of the application in such cases depends on the implementation.
XML Schema Samples in the XDK

Example 7–3 shows a sample XML document that contains a purchase report that describes the parts that have been ordered in different regions. This sample file is located at $ORACLE_HOME/xdk/demo/java/schema/report.xml.

Example 7–3  report.xml

```xml
<purchaseReport
    xmlns="http://www.example.com/Report"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.example.com/Report  report.xsd"
    period="P3M" periodEnding='1999-12-31'>
  <regions>
    <zip code="95819">
      <part number="872-AA" quantity='1'/>
      <part number="926-AA" quantity='1'/>
      <part number="833-AA" quantity='1'/>
      <part number="455-BX" quantity='1'/>
    </zip>
    <zip code="63143">
      <part number="455-BX" quantity='4'/>
    </zip>
  </regions>
  <parts>
    <part number="872-AA">Lawnmower</part>
    <part number="926-AA">Baby Monitor</part>
    <part number="833-AA">Lapis Necklace</part>
    <part number="455-BX">Sturdy Shelves</part>
  </parts>
</purchaseReport>
```

Example 7–4 shows the XML schema document named report.xsd, which is the sample XML schema document that you can use to validate report.xml. Among other things, the XML schema defines the names of the elements that are legal in the instance document as well as the type of data that the elements can contain.

Example 7–4  report.xsd

```xml
<schema targetNamespace="http://www.example.com/Report"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:r="http://www.example.com/Report"
    elementFormDefault='qualified'>

  <annotation>
    <documentation xml:lang='en'>
      Report schema for Example.com
      Copyright 2000 Example.com. All rights reserved.
    </documentation>
  </annotation>

  <element name="purchaseReport">
    <complexType>
      <sequence>
        <keyref name="dummy2" refer="r:pNumKey">
          <selector xpath="/r:zip/r:part"/>
          <field xpath="/number"/>
        </keyref>
      </sequence>
    </complexType>
  </element>

</schema>
```
<element name="parts" type="r:PartsType"/>
</sequence>
<attribute name="period" type="duration"/>
<attribute name="periodEnding" type="date"/>
</complexType>

<unique name="dummy1">
<selector xpath="r:regions/r:zip"/>
<field xpath="@code"/>
</unique>

?key name="pNumKey">
<selector xpath="r:parts/r:part"/>
<field xpath="@number"/>
</key>
</element>
<complexType name="RegionsType">
<sequence>
<element name="zip" maxOccurs="unbounded">
<complexType>
<sequence>
<element name="part" maxOccurs="unbounded">
<complexType>
<complexContent>
<restriction base="anyType">
<attribute name="number" type="r:SKU"/>
<attribute name="quantity" type="positiveInteger"/>
</restriction>
</complexContent>
</complexType>
</element>
</sequence>
<attribute name="code" type="positiveInteger"/>
</complexType>
</element>
</sequence>
<attribute name="code" type="positiveInteger"/>
</complexType>
</element>
</sequence>
</complexType>

<complexType name="PartsType">
<sequence>
<element name="part" maxOccurs="unbounded">
<complexType>
<simpleContent>
<extension base="string">
<attribute name="number" type="r:SKU"/>
</extension>
</simpleContent>
</complexType>
</element>
</sequence>
</complexType>

<complexType name="SKU">
<restriction base="string">
<pattern value="\d{3}-(A-Z){2}"/>
</restriction>
</complexType>

</schema>
Differences Between XML Schemas and DTDs

The XML Schema language includes most of the capabilities of the DTD specification. An XML schema serves a similar purpose to a DTD, but is more flexible in specifying document constraints. Table 7–1 compares some of the features between the two validation mechanisms.

Table 7–1  Feature Comparison Between XML Schema and DTD

<table>
<thead>
<tr>
<th>Feature</th>
<th>XML Schema</th>
<th>DTD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element nesting</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Element occurrence constraints</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Permitted attributes</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Attribute types and default values</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Written in XML</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Namespace support</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Built-In datatypes</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>User-Defined datatypes</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Include/Import</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Refinement (inheritance)</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

The following reasons are probably the most persuasive for choosing XML schema validation over DTD validation:

- The XML Schema language enables you to define rules for the content of elements and attributes. You achieve control over content by using datatypes. With XML Schema datatypes you can more easily perform actions such as the following:
  - Declare which elements should contain which types of data, for example, positive integers in one element and years in another
  - Process data obtained from a database
  - Define restrictions on data, for example, a number between 10 and 20
  - Define data formats, for example, dates in the form MM-DD-YYYY
  - Convert data between different datatypes, for example, strings to dates

- Unlike DTD grammar, documents written in the XML Schema language are themselves written in XML. Thus, you can perform the following actions:
  - Use your XML parser to parse your XML schema
  - Process your XML schema with the XML DOM
  - Transform your XML document with XSLT
  - Reuse your XML schemas in other XML schemas
  - Extend your XML schema by adding elements and attributes
  - Reference multiple XML schemas from the same document
Using the XML Schema Processor: Overview

The Oracle XML Schema processor is a SAX-based XML schema validator that you can use to validate instance documents against an XML schema. The processor supports both LAX and strict validation.

You can use the processor in the following ways:

- Enable it in the XML parser
- Use it with a DOM tree to validate whole or part of an XML document
- Use it as a component in a processing pipeline (like a content handler)

You can configure the schema processor in different ways depending on your requirements. For example, you can do the following:

- Use a fixed XML schema or automatically build a schema based on the schemaLocation attributes in an instance document.
- Set XMLError and entityResolver to gain better control over the validation process.
- Determine how much of an instance document should be validated. You can use any of the validation modes specified in Table 4–1. You can also designate a type of element as the root of validation.

Using the XML Schema Processor: Basic Process

The following XDK packages are important for applications that process XML schemas:

- oracle.xml.parser.v2, which provides APIs for XML parsing
- oracle.xml.parser.schema, which provides APIs for XML Schema processing

The most important classes in the oracle.xml.parser.schema package are described in Table 7–2. These form the core of most XML schema applications.

Table 7–2 ore.xm...
Using the XML Schema Processor: Overview

Table 7-2 (Cont.) oracle.xml.parser.schema Classes

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Description</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLSchemaNode class</td>
<td>Represents schema components in a target namespace, including type definitions, element and attribute declarations, and group and attribute group definitions.</td>
<td>The principal methods are get methods such as <code>getElementSet()</code> and <code>getAttributeDeclarations()</code> obtain components of the XML schema.</td>
</tr>
</tbody>
</table>
| XSDBuilder class    | Builds an XMLSchema object from an XML schema document. The XMLSchema object is a set of objects (Infoset items) corresponding to top-level schema declarations and definitions. The schema document is XML parsed and converted to a DOM tree. | The principal methods are as follows:  
  - `build()` creates an XMLSchema object.  
  - `getObject()` returns the XMLSchema object.  
  - `setEntityResolver()` sets an EntityResolver for resolving imports and includes. |
| XSDValidator class  | Validates an instance XML document against an XML schema. When registered, an XSDValidator object is inserted as a pipeline node between XMLParser and XMLDocument events handlers. | The principal methods are as follows:  
  - `get` methods such as `getCurrentMode()` and `getElementDeclaration()`  
  - `set` methods such as `setXMLProperty()` and `setDocumentLocator()`  
  - `startDocument()` receives notification of the beginning of the document.  
  - `startElement()` receives notification of the beginning of the element. |

Figure 7–1 depicts the basic process of validating an instance document with the XML Schema processor.

Figure 7–1 XML Schema Processor

![XML Schema Processor Diagram]

The XML Schema processor performs the following major tasks:

1. A builder (XSDBuilder object) assembles the XML schema from an input XML schema document. Although instance documents and schemas need not exist specifically as files on the operating system, they are commonly referred to as files. They may exist as streams of bytes, fields in a database record, or collections of XML Infoset "Information Items."
This task involves parsing the schema document into an object. The builder creates the schema object explicitly or implicitly:

- In explicit mode, you pass in an XML schema when you invoke the processor. "Validating Against Externally Referenced XML Schemas" on page 7-13 explains how to build the schema object in explicit mode.

- In implicit mode, you do not pass in an XML schema when you invoke the processor because the schema is internally referenced by the instance document. "Validating Against Internally Referenced XML Schemas" on page 7-12 explains how to create the schema object in implicit mode.

2. The XML schema validator uses the schema object to validate the instance document. This task involves the following steps:

   a. A SAX parser parses the instance document into SAX events, which it passes to the validator.

   b. The validator receives SAX events as input and validates them against the schema object, sending an error message if it finds invalid XML components. "Validation in the XML Parser" on page 4-8 describes the validation modes that you can use when validating the instance document. If you do not explicitly set a schema for validation with the XSDBuilder class, then the instance document must have the correct xsi:schemaLocation attribute pointing to the schema file. Otherwise, the program will not perform the validation. If the processor encounters errors, it generates error messages.

   c. The validator sends input SAX events, default values, or post-schema validation information to a DOM builder or application.

See Also:

- Oracle Database XML Java API Reference to learn about the XSDBuilder, DOMParser, and SAXParser classes
- Chapter 7, “Using the Schema Processor for Java” to learn about the XDK SAX and DOM parsers

Running the XML Schema Processor Demo Programs

Demo programs for the XML Schema processor for Java are included in $ORACLE_HOME/xdk/demo/java/schema. Table 7–3 describes the XML files and programs that you can use to test the XML Schema processor.

Table 7–3 XML Schema Sample Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat.xsd</td>
<td>A sample XML schema used by the XSDSetSchema.java program to validate catalogue.xml. The cat.xsd schema specifies the structure of a catalogue of books.</td>
</tr>
<tr>
<td>catalogue.xml</td>
<td>A sample instance document that the XSDSetSchema.java program uses to validate against the cat.xsd schema.</td>
</tr>
<tr>
<td>catalogue_e.xml</td>
<td>A sample instance document used by the XSDSample.java program. When the program tries to validate this document against the cat.xsd schema, it generates schema errors.</td>
</tr>
<tr>
<td>DTD2Schema.java</td>
<td>This sample program converts a DTD (first argument) into an XML Schema and uses it to validate an XML file (second argument).</td>
</tr>
<tr>
<td>embeded_xsql.xsd</td>
<td>The XML schema used by XSDLax.java. The schema defines the structure of an XSQL page.</td>
</tr>
<tr>
<td>embeded_xsql.xml</td>
<td>The instance document used by XSDLax.java.</td>
</tr>
</tbody>
</table>
Table 7–3 (Cont.) XML Schema Sample Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>juicer1.xml</td>
<td>A sample XML document for use with xsdproperty.java. The XML schema that defines this document is juicer1.xsd.</td>
</tr>
<tr>
<td>juicer1.xsd</td>
<td>A sample XML schema for use with xsdproperty.java. This XML schema defines juicer1.xml.</td>
</tr>
<tr>
<td>juicer2.xml</td>
<td>A sample XML document for use with xsdproperty.java. The XML schema that defines this document is juicer2.xsd.</td>
</tr>
<tr>
<td>juicer2.xsd</td>
<td>A sample XML document for use with xsdproperty.java. This XML schema defines juicer2.xml.</td>
</tr>
<tr>
<td>report.xml</td>
<td>The sample XML file that XSDSetSchema.java uses to validate against the XML schema report.xsd.</td>
</tr>
<tr>
<td>report.xsd</td>
<td>A sample XML schema used by the XSDSetSchema.java program to validate the contents of report.xml. The report.xsd schema specifies the structure of a purchase order.</td>
</tr>
<tr>
<td>report_e.xml</td>
<td>When the program validates this sample XML file using XSDSample.java, it generates XML Schema errors.</td>
</tr>
<tr>
<td>xsddom.java</td>
<td>This program shows how to validate an instance document by obtain a DOM representation of the document and using an XSDValidator object to validate it.</td>
</tr>
<tr>
<td>xsdent.java</td>
<td>This program validates an XML document by redirecting the referenced schema in the SchemaLocation attribute to a local version.</td>
</tr>
<tr>
<td>xsdent.xml</td>
<td>This XML document describes a book. The file is used as an input to xsdent.java.</td>
</tr>
<tr>
<td>xsdent.xsd</td>
<td>This XML schema document defines the rules for xsdent.xml. The schema document contains a schemaLocation attribute set to xsdent-1.xsd.</td>
</tr>
<tr>
<td>xsdent-1.xsd</td>
<td>The XML schema document referenced by the schemaLocation attribute in xsdent.xsd.</td>
</tr>
<tr>
<td>xsdproperty.java</td>
<td>This demo shows how to configure the XML Schema processor to validate an XML document based on a complex type or element declaration.</td>
</tr>
<tr>
<td>xsdsax.java</td>
<td>This demo shows how to validate an XML document received as a SAX stream.</td>
</tr>
<tr>
<td>XSDLax.java</td>
<td>This demo is the same as XSDSetSchema.java but sets the SCHEMA_LAX_VALIDATION flag for LAX validation.</td>
</tr>
<tr>
<td>XSDSample.java</td>
<td>This program is a sample driver that you can use to process XML instance documents.</td>
</tr>
<tr>
<td>XSDSetSchema.java</td>
<td>This program is a sample driver to process XML instance documents by overriding the schemaLocation. The program uses the XML Schema specification from cat.xsd to validate the contents of catalogue.xml.</td>
</tr>
</tbody>
</table>

Documentation for how to compile and run the sample programs is located in the README in the same directory. The basic steps are as follows:

1. Change into the $ORACLE_HOME/xdk/demo/java/schema directory (UNIX) or %ORACLE_HOME%\xdk\demo\java\schema directory (Windows).
2. Run make (UNIX) or Make.bat (Windows) at the command line.
3. Add xmlparserv2.jar, xschema.jar, and the current directory to the CLASSPATH. These JAR files are located in $ORACLE_HOME/1ib (UNIX) and %ORACLE_HOME%\lib (Windows). For example, you can set the CLASSPATH as follows with the tcsh shell on UNIX:

   ```bash
   setenv CLASSPATH "$CLASSPATH:\$ORACLE_HOME\lib\xmlparserv2.jar;$ORACLE_HOME\lib\schema.jar:.
   ```

   Note that the XML Schema processor requires JDK version 1.2 or higher and is usable on any operating system with Java 1.2 support.
4. Run the sample programs with the XML files that are included in the directory:

   - The following examples use `report.xsd` to validate the contents of `report.xml`:
     
     ```
     java XSDSample report.xml
     java XSDSetSchema report.xsd report.xml
     ```

   - The following example validates an instance document in Lax mode:
     
     ```
     java XSDLax embeded_xsql.xsd embeded_xsql.xml
     ```

   - The following examples use `cat.xsd` to validate the contents of `catalogue.xml`:
     
     ```
     java XSDSample catalogue.xml
     java XSDSetSchema cat.xsd catalogue.xml
     ```

   - The following examples generates error messages:
     
     ```
     java XSDSample catalogue_e.xml
     java XSDSample report_e.xml
     ```

   - The following example uses the `schemaLocation` attribute in `xsdent.xsd` to redirect the XML schema to `xsdent-1.xsd` for validation:
     
     ```
     java xsdent xsdent.xml xsdent.xsd
     ```

   - The following example generates a SAX stream from `report.xml` and validates it against the XML schema defined in `report.xsd`:
     
     ```
     java xsdsax report.xsd report.xml
     ```

   - The following example creates a DOM representation of `report.xml` and validates it against the XML schema defined in `report.xsd`:
     
     ```
     java xsddom report.xsd report.xml
     ```

   - The following examples configure validation starting with an element declaration or complex type definition:
     
     ```
     java xsdproperty juicer1.xml juicer1.xsd http://www.juicers.org \ juicersType false > juicersType.out
     java xsdproperty juicer2.xml juicer2.xsd http://www.juicers.org \ Juicers true > juicers_e.out
     ```

   - The following example converts a DTD (`dtd2schema.dtd`) into an XML schema and uses it to validate an instance document (`dtd2schema.xml`):
     
     ```
     java DTD2Schema dtd2schema.dtd dtd2schema.xml
     ```

Using the XML Schema Processor Command-Line Utility

"Using the XML Parser Command-Line Utility" on page 4-13 describes how to run the `oraxml` command-line utility. You can use this utility to validate instance documents against XML schemas and DTDs.

Using `oraxml` to Validate Against a Schema

Change into the `$ORACLE_HOME/xdk/demo/java/schema` directory. Example 7–5 shows how you can validate `report.xml` against `report.xsd` by executing the following on the command line.
Validating XML with XML Schemas

Example 7–5  Using oraxml to Validate Against a Schema

oraxml -schema -enc report.xml

You should obtain the following output:

The encoding of the input file: UTF-8
The input XML file is parsed without errors using Schema validation mode.

Using oraxml to Validate Against a DTD

Change into the $ORACLE_HOME/xdk/demo/java/parser/common directory. Example 7–6 shows how you can validate family.xml against family.dtd by executing the following on the command line.

Example 7–6  Using oraxml to Validate Against a DTD

oraxml -dtd -enc family.xml

You should obtain the following output:

The encoding of the input file: UTF-8
The input XML file is parsed without errors using DTD validation mode.

Validating XML with XML Schemas

This section includes the following topics:

■ Validating Against Internally Referenced XML Schemas
■ Validating Against Externally Referenced XML Schemas
■ Validating a Subsection of an XML Document
■ Validating XML from a SAX Stream
■ Validating XML from a DOM
■ Validating XML from Designed Types and Elements
■ Validating XML with the XSDValidator Bean

Validating Against Internally Referenced XML Schemas

The $ORACLE_HOME/xdk/demo/java/schema/XSDSample.java program illustrates how to validate against an implicit XML Schema. The validation mode is implicit because the XML schema is referenced in the instance document itself.

Follow the steps in this section to write programs that use the setValidationMode() method of the oracle.xml.parser.v2.DOMParser class:

1. Create a DOM parser to use for the validation of an instance document. The following code fragment from XSDSample.java illustrates how to create the DOMParser object:

   ```java
   public class XSDSample
   {
       public static void main(String[] args) throws Exception
       {
           if (args.length != 1)
           {
               System.out.println("Usage: java XSDSample <filename>");
               return;
           }
   ```
Validating XML with XML Schemas

Using the Schema Processor for Java

7-13

```
process (args[0]);
}

public static void process (String xmlURI) throws Exception
{
    DOMParser dp = new DOMParser();
    URL url = createURL(xmlURI);
    ...
}
...
```

createURL() is a helper method that constructs a URL from a filename passed to the program as an argument.

2. Set the validation mode for the validating DOM parser with the DOMParser.setValidationMode() method. For example, XSDSample.java shows how to specify XML schema validation:

```
dp.setValidationMode(DOMParser.SCHEMA_VALIDATION);
dp.setPreserveWhitespace(true);
```

3. Set the output error stream with the DOMParser.setErrorStream() method. For example, XSDSample.java sets the error stream for the DOM parser object as follows:

```
dp.setErrorStream(System.out);
```

4. Validate the instance document with the DOMParser.parse() method. You do not have to create an XML schema object explicitly because the schema is internally referenced by the instance document. For example, XSDSample.java validates the instance document as follows:

```
try
{
    System.out.println("Parsing *+xmlURI");
    dp.parse(url);
    System.out.println("The input file *+xmlURI"+ parsed without errors");
}
catch (XMLParseException pe)
{
    System.out.println("Parser Exception: ' + pe.getMessage());
}
catch (Exception e)
{
    System.out.println("NonParserException: ' + e.getMessage());
}
```

Validating Against Externally Referenced XML Schemas

The $ORACLE_HOME/xdk/demo/java/schema/XSDSetSchema.java program illustrates how to validate an XML schema explicitly. The validation mode is explicit because you use the XSDBuilder class to specify the schema to use for validation: the schema is not specified in the instance document as in implicit validation.

Follow the basic steps in this section to write Java programs that use the build() method of the oracle.xml.parser.schema.XSDBuilder class:

1. Build an XML schema object from the XML schema document with the XSDBuilder.build() method. The following code fragment from XSDSetSchema.java illustrates how to create the object:

```
Validating Against Externally Referenced XML Schemas

The $ORACLE_HOME/xdk/demo/java/schema/XSDSetSchema.java program illustrates how to validate an XML schema explicitly. The validation mode is explicit because you use the XSDBuilder class to specify the schema to use for validation: the schema is not specified in the instance document as in implicit validation.

Follow the basic steps in this section to write Java programs that use the build() method of the oracle.xml.parser.schema.XSDBuilder class:

1. Build an XML schema object from the XML schema document with the XSDBuilder.build() method. The following code fragment from XSDSetSchema.java illustrates how to create the object:
public class XSDSetSchema
{
    public static void main(String[] args) throws Exception
    {
        if (args.length != 2)
        {
            System.out.println("Usage: java XSDSample <schema_file> <xml_file>*");
            return;
        }
        XSDBuilder builder = new XSDBuilder();
        URL    url =  createURL(args[0]);
        // Build XML Schema Object
        XMLSchema schemadoc = (XMLSchema)builder.build(url);
        process(args[1], schemadoc);
    }
}

The `createURL()` method is a helper method that constructs a URL from the schema document filename specified on the command line.

2. Create a DOM parser to use for validation of the instance document. The following code from `XSDSetSchema.java` illustrates how to pass the instance document filename and XML schema object to the `process()` method:

    public static void process(String xmlURI, XMLSchema schemadoc)
    throws Exception
    {
        DOMParser dp  = new DOMParser();
        URL       url = createURL (xmlURI);
        ...
    }

3. Specify the schema object to use for validation with the `DOMParser.setXMLSchema()` method. This step is not necessary in implicit validation mode because the instance document already references the schema. For example, `XSDSetSchema.java` specifies the schema as follows:

    dp.setXMLSchema(schemadoc);

4. Set the validation mode for the DOM parser object with the `DOMParser.setValidationMode()` method. For example, `XSDSample.java` shows how to specify XML schema validation:

    dp.setValidationMode(XMLParser.SCHEMA_VALIDATION);
    dp.setPreserveWhitespace(true);

5. Set the output error stream for the parser with the `DOMParser.setErrorStream()` method. For example, `XSDSetSchema.java` sets it as follows:

    dp.setErrorStream (System.out);

6. Validate the instance document against the XML schema with the `DOMParser.parse()` method. For example, `XSDSetSchema.java` includes the following code:

    try
    {
        System.out.println("Parsing "+xmlURI);
        dp.parse (url);
        System.out.println("The input file "+xmlURI+" parsed without errors");
Validating a Subsection of an XML Document

In LAX mode, you can validate parts of the XML content of an instance document without validating the whole document. A LAX parser indicates that the processor should perform validation for elements in the instance document that are declared in an associated XML schema. The processor does not consider the instance document invalid if it contains no elements declared in the schema.

By using LAX mode, you can define the schema only for the part of the XML that you want to validate. The $ORACLE_HOME/xdk/demo/java/schema/XSDLax.java program illustrates how to use LAX validation. The program follows the basic steps described in "Validating Against Externally Referenced XML Schemas" on page 7-13:

1. Build an XML schema object from the user-specified XML schema document.
2. Create a DOM parser to use for validation of the instance document.
3. Specify the XML schema to use for validation.
4. Set the validation mode for the DOM parser object.
5. Set the output error stream for the parser.
6. Validate the instance document against the XML schema by calling DOMParser.parse().

To enable LAX validation, the program sets the validation mode in the parser to SCHEMA_LAX_VALIDATION rather than to SCHEMA_VALIDATION. The following code fragment from XSDLax.java illustrates this technique:

```java
dp.setXMLSchema(schemadoc);
dp.setValidationMode(XMLParser.SCHEMA_LAX_VALIDATION);
dp.setPreserveWhitespace (true);
```

You can test LAX validation by running the sample program as follows:
```
java XSDLax embeded_xsql.xsd embeded_xsql.xml
```

Validating XML from a SAX Stream

The $ORACLE_HOME/xdk/demo/java/schema/xsdsax.java program illustrates how to validate an XML document received as a SAX stream. You instantiate an XSDValidator and register it with the SAX parser as the content handler.

Follow the steps in this section to write programs that validate XML from a SAX stream:

1. Build an XML schema object from the user-specified XML schema document by invoking the XSDBuilder.build() method. The following code fragment from XSDLax.java illustrates how to create the object:

```java
XSDBuilder builder = new XSDBuilder();
```
URL url = XMLUtil.createURL(args[0]);

// Build XML Schema Object
XMLSchema schemadoc = (XMLSchema)builder.build(url);
process(args[1], schemadoc);
...

createURL() is a helper method that constructs a URL from the filename specified on the command line.

2. Create a SAX parser (SAXParser object) to use for validation of the instance document. The following code fragment from saxxsd.java passes the handles to the XML document and schema document to the process() method:

   process(args[1], schemadoc);
   ...

   public static void process(String xmlURI, XMLSchema schemadoc)
   throws Exception
   {
      SAXParser dp = new SAXParser();
      ...

3. Configure the SAX parser. The following code fragment sets the validation mode for the SAX parser object with the XSDBuilder.setValidationMode() method:

   dp.setPreserveWhitespace (true);
   dp.setValidationMode(XMLParser.NONVALIDATING);

4. Create and configure a validator (XSDValidator object). The following code fragment illustrates this technique:

   XMLError err;  
   ...
   err = new XMLError();
   ...
   XSDValidator validator = new XSDValidator();
   ...
   validator.setError(err);

5. Specify the XML schema to use for validation by invoking the XSDBuilder.setXMLProperty() method. The first argument is the name of the property, which is fixedSchema, and the second is the reference to the XML schema object. The following code fragment illustrates this technique:

   validator.setXMLProperty(XSDNode.FIXED_SCHEMA, schemadoc);
   ...

6. Register the validator as the SAX content handler for the parser. The following code fragment illustrates this technique:

   dp.setContentHandler(validator);
   ...

7. Validate the instance document against the XML schema by invoking the SAXParser.parse() method. The following code fragment illustrates this technique:

   dp.parse(url);

Validate XML with XML Schemas
Validating XML from a DOM

The $ORACLE_HOME/xdk/demo/java/schema/xsddom.java program shows how to validate an instance document by obtain a DOM representation of the document and using an XSDValidator object to validate it.

The xsddom.java program follows these steps:

1. Build an XML schema object from the user-specified XML schema document by invoking the XSDBuilder.build() method. The following code fragment from illustrates how to create the object:

   XSDBuilder builder = new XSDBuilder();
   URL    url = XMLUtil.createURL(args[0]);
   XMLSchema schemadoc = (XMLSchema)builder.build(url);
   process(args[1], schemadoc);

   createURL() is a helper method that constructs a URL from the filename specified on the command line.

2. Create a DOM parser (DOMParser object) to use for validation of the instance document. The following code fragment from domxsd.java passes the handles to the XML document and schema document to the process() method:

   process(args[1], schemadoc);
   ...
   public static void process(String xmlURI, XMLSchema schemadoc)
   throws Exception
   {
       DOMParser dp  = new DOMParser();
       ...
   }

3. Configure the DOM parser. The following code fragment sets the validation mode for the parser object with the DOMParser.setValidationMode() method:

   dp.setPreserveWhitespace (true);
   dp.setValidationMode(XMLParser.NONVALIDATING);
   dp.setErrorStream (System.out);

4. Parse the instance document. The following code fragment illustrates this technique:

   dp.parse (url);

5. Obtain the DOM representation of the input document. The following code fragment illustrates this technique:

   XMLDocument doc = dp.getDocument();

6. Create and configure a validator (XSDValidator object). The following code fragment illustrates this technique:

   XMLError err;
   ...
   err = new XMLError();
   ...
   XSDValidator validator = new XSDValidator();
   ...
   validator.setError(err);
7. Specify the schema object to use for validation by invoking the
   XSDBuilder.setXMLProperty() method. The first argument is the name of the
   property, which in this example is fixedSchema, and the second is the reference to
   the schema object. The following code fragment illustrates this technique:

   validator.setXMLProperty(XSDNode.FIXED_SCHEMA, schemadoc);
   ...

8. Obtain the root element (XMLElement) of the DOM tree and validate. The following
   code fragment illustrates this technique:

   XMLElement root = (XMLElement)doc.getDocumentElement();
   XMLElement copy = (XMLElement)root.validateContent(validator, true);
   copy.print(System.out);

Validating XML from Designed Types and Elements

The $ORACLE_HOME/xdk/demo/java/schema/xsdproperty.java program shows how to
configure the XML Schema processor to validate an XML document based on a
complex type or element declaration.

The xsdproperty.java program follows these steps:

1. Create String objects for the instance document name, XML schema name, root
   node namespace, root node local name, and specification of element or complex
type ("true" means the root node is an element declaration). The following code
fragment illustrates this technique:

   String xmlfile = args[0];
   String xsdfile = args[1];
   ...
   String ns = args[2]; //namespace for the root node
   String nm = args[3]; //root node's local name
   String el = args[4]; //true if root node is element declaration,
   // otherwise, the root node is a complex type

2. Create an XSD builder and use it to create the schema object. The following code
fragment illustrates this technique:

   XSDBuilder builder = new XSDBuilder();
   URL url = XMLUtil.createURL(xsdfile);
   XMLSchema schema;
   ...
   schema = (XMLSchema) builder.build(url);

3. Obtain the node. Invoke different methods depending on whether the node is an
   element declaration or a complex type:

   ■ If the node is an element declaration, pass the local name and namespace to
     the getElement() method of the schema object.
   ■ If the node is an element declaration, pass the namespace, local name, and root
     complex type to the getType() method of the schema object.

   xsdproperty.java uses the following control structure:

   QName qname = new QName(ns, nm);
   ...
   XSDNode nd;
   ...
   if (el.equals("true"))
   {

Validating XML with XML Schemas

Using the Schema Processor for Java

4. After obtaining the node, create a new parser and set the schema to the parser to enable schema validation. The following code fragment illustrates this technique:

```java
DOMParser dp = new DOMParser();
URL url = XMLUtil.createURL (xmlURI);
```

5. Set properties on the parser and then parse the URL. Invoke the `schemaValidatorProperty()` method as follows:

   a. Set the root element or type property on the parser to a fully qualified name.

   For a top-level element declaration, set the property name to `XSDNode.ROOT_ELEMENT` and the value to a `QName`, as illustrated by the `process1()` method.

   For a top-level type definition, set the property name to `XSDNode.ROOT_TYPE` and the value to a `QName`, as illustrated by the `process2()` method.

   b. Set the root node property on the parser to an element or complex type node.

   For an element node, set the property name to `XSDNode.ROOT_NODE` and the value to an `XSDElement` node, as illustrated by the `process3()` method.

   For a type node, set the property name to `XSDNode.ROOT_NODE` and the value to an `XSDComplexType` node, as illustrated by the `process3()` method.

The following code fragment shows the sequence of method invocation:

```java
if (el.equals("true"))
{
    nd = schema.getElement(ns, nm);
    process1(xmlfile, schema, qname);
    process3(xmlfile, schema, nd);
}
else
{
    nd = schema.getType(ns, nm, XSDNode.TYPE);
    process2(xmlfile, schema, qname);
    process3(xmlfile, schema, nd);
}
```

The processing methods are implemented as follows:

```java
static void process1(String xmlURI, XMLSchema schema, QName qname)
   throws Exception
{
    /* create parser... */
    dp.setXMLSchema(schema);
    dp.setSchemaValidatorProperty(XSDNode.ROOT_ELEMENT, qname);
    dp.setPreserveWhitespace (true);
    dp.setErrorStream (System.out);
    dp.parse (url);
    ...
}
```
static void process2(String xmlURI, XMLSchema schema, QxName qname)
    throws Exception
{
    /* create parser... */
    dp.setXMLSchema(schema);
    dp.setSchemaValidatorProperty(XSDNode.ROOT_TYPE, qname);
    dp.setPreserveWhitespace (true);
    dp.setErrorStream (System.out);
    dp.parse (url);
    ...
}

static void process3(String xmlURI, XMLSchema schema, XSDNode node)
    throws Exception
{
    /* create parser... */

    dp.setXMLSchema(schema);
    dp.setSchemaValidatorProperty(XSDNode.ROOT_NODE, node);
    dp.setPreserveWhitespace (true);
    dp.setErrorStream (System.out);
    dp.parse (url);
    ...
}

Validating XML with the XSDValidator Bean

The oracle.xml.schemavalidator.XSDValidator bean encapsulates the
oracle.xml.parser.schema.XSDValidator class and adds functionality for validating
a DOM tree. The parser builds the DOM tree for the instance document and XML
schema document and validates the instance document against the schema.

The XSDValidatorSample.java program in $ORACLE_HOME/xdk/demo/java/transviewer illustrates how to use the XSDValidator bean.

Follow the basic steps in this section to write Java programs that use the XSDValidator bean:

1. Parse the instance document with the DOMParser.parse() method. The following
code fragment from XSDValidatorSample.java illustrates this technique:

   URL xmlinstanceurl, schemaurl;
   XMLDocument xmldoc1,xmldoc2;

   // get the URL for the input files
   xmlinstanceurl = createURL(args[0]);
   // Parse the XML Instance document first
   xmldoc1 = parseXMLDocument(xmlinstanceurl);

   createURL() is a helper method that creates a URL from a filename. The
   parseXMLDocument() method receives a URL as input and parses it with the
   DOMParser.parse() method as follows:
   DOMParser parser = new DOMParser();
   parser.parse(xmlurl);
   return parser.getDocument();

2. Parse the XML schema document with the DOMParser.parse() method. The
following code from XSDValidatorSample.java illustrates this technique:

   schemaurl = createURL(args[1]);
Tips and Techniques for Programming with XML Schemas

This section contains the following topics:

- Overriding the Schema Location with an Entity Resolver
- Converting DTDs to XML Schemas

Overriding the Schema Location with an Entity Resolver

When the XSDBuilder builds a schema, it may need to include or import other schemas specified as URLs in the schemaLocation attribute. The xsdent.java demo described in Table 7–3 illustrates this case. The document element in xsdent.xml file contains the following attribute:

```xml
xsi:schemaLocation =  "http://www.example.com/BookCatalogue
xsdent.xsd">
```

The xsdent.xsd document contains the following elements:

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.com/BookCatalogue"
    xmlns:catd = "http://www.example.com/Digest"
```
xmlns:cat = 'http://www.example.com/BookCatalogue'
elementFormDefault="qualified">
<import namespace='http://www.example.com/Digest'
schemaLocation='xsdent-1.xsd' />

In some cases, you may want to override the schema locations specified in <import> and supply the builder with the required schema documents. For example, you may have downloaded the schemas documents from external Web sites and stored them in a database. In such cases, you can set an entity resolver in the XSDBuilder. XSDBuilder passes the schema location to the resolver, which returns an InputStream, Reader, or URL as an InputSource. The builder can read the schema documents from the InputSource.

The xsdent.java program illustrates how you can override the schema location with an entity resolver. You must implement the EntityResolver interface, instantiate the entity resolver, and set it in the XML schema builder. In the demo code, sampleEntityResolver1 returns InputSource as an InputStream whereas sampleEntityResolver2 returns InputSource as a URL.

Follow these basic steps:

1. Create a new XML schema builder as follows:
   ```java
   XSDBuilder builder = new XSDBuilder();
   ```

2. Set the builder to your entity resolver. An entity resolver is a class that implements the EntityResolver interface. The purpose of the resolver is to enable the XML reader to intercept any external entities before including them. The following code fragment creates an entity resolver and sets it in the builder:
   ```java
   builder.setEntityResolver(new sampleEntityResolver1());
   ```

   The sampleEntityResolver1 class implements the resolveEntity() method. You can use this method to redirect external system identifiers to local URIs. The source code is as follows:
   ```java
   class sampleEntityResolver1 implements EntityResolver {
    public InputSource resolveEntity (String targetNS, String systemId) throws SAXException, IOException {
    // perform any validation check if needed based on targetNS & systemId
    InputSource mySource = null;
    URL u = XMLUtil.createURL(systemId);
    // Create input source with InputStream as input
    mySource = new InputSource(u.openStream());
    mySource.setSystemId(systemId);
    return mySource;
    }
   }
   ```

   Note that sampleEntityResolver1 initializes the InputSource with a stream.

3. Build the XML schema object. The following code illustrates this technique:
   ```java
   schemadoc = builder.build(url);  
   ```

4. Validate the instance document against the XML schema. The program executes the following statement:
   ```java
   process(xmlfile, schemadoc);
   ```
The `process()` method creates a DOM parser, configures it, and invokes the `parse()` method. The method is implemented as follows:

```java
public static void process(String xmlURI, Object schemadoc)
  throws Exception
{
    DOMParser dp = new DOMParser();
    URL url = XMLUtil.createURL(xmlURI);

    dp.setXMLSchema(schemadoc);
    dp.setValidationMode(XMLParser.SCHEMA_VALIDATION);
    dp.setPreserveWhitespace (true);
    dp.setErrorStream (System.out);
    try {
      dp.parse(url);
      ...
    }
}
```

Converting DTDs to XML Schemas

Because of the power and flexibility of the XML Schema language, you may want to convert your existing DTDs to XML Schema documents. The XDK API enables you to perform this transformation.

The `$ORACLE_HOME/xdk/demo/java/schema/DTD2Schema.java` program illustrates how to convert a DTD. You can test the program as follows:

```
java DTD2Schema dtd2schema.dtd dtd2schema.xml
```

Follow these basic steps to convert a DTD to an XML schema document:

1. Parse the DTD with the `DOMParser.parseDTD()` method. The following code fragment from `DTD2Schema.java` illustrates how to create the DTD object:

   ```java
   XSDBuilder builder = new XSDBuilder();
   URL dtdURL = createURL(args[0]);
   DTD dtd = getDTD(dtdURL, "abc");
   ```

   The `getDTD()` method is implemented as follows:

   ```java
   private static DTD getDTD(URL dtdURL, String rootName)
     throws Exception
   {
     DOMParser parser = new DOMParser();
     DTD dtd;
     parser.setValidationMode(true);
     parser.setErrorStream(System.out);
     parser.showWarnings(true);
     parser.parseDTD(dtdURL, rootName);
     dtd = (DTD)parser.getDoctype();
     return dtd;
   }
   ```

2. Convert the DTD to an XML schema DOM tree with the `DTD.convertDTD2Sdhema()` method. The following code fragment from `DTD2Schema.java` illustrates this technique:

   ```java
   XMLDocument dtddoc = dtd.convertDTD2Schema();
   ```

3. Write the XML schema DOM tree to an output stream with the `XMLDocument.print()` method. The following code fragment from `DTD2Schema.java` illustrates this technique:
4. Create an XML schema object from the schema DOM tree with the `XSDBuilder.build()` method. The following code fragment from `DTD2Schema.java` illustrates this technique:

```java
XMLSchema schemadoc = (XMLSchema)builder.build(dtddoc, null);
```

5. Validate an instance document against the XML schema with the `DOMParser.parse()` method. The following code fragment from `DTD2Schema.java` illustrates this technique:

```java
validate(args[1], schemadoc);
```

The `validate()` method is implemented as follows:

```java
DOMParser dp = new DOMParser();
URL url = createURL (xmlURI);
dp.setXMLSchema(schemadoc);
dp.setValidationMode(XMLParser.SCHEMA_VALIDATION);
dp.setPreserveWhitespace (true);
dp.setErrorStream (System.out);
try {
    System.out.println("Parsing "+xmlURI);
    dp.parse (url);
    System.out.println("The input file <"+xmlURI+"> parsed without errors");
} 
...
```
Using the JAXB Class Generator

This chapter contains the following topics:

- Introduction to the JAXB Class Generator
- Using the JAXB Class Generator: Overview
- Processing XML with the JAXB Class Generator

**Note:** Use the JAXB class generator for new applications in order to use the object binding feature for XML data. The Oracle9i class generator for Java is deprecated. Oracle Database 10g supports the Oracle9i class generator for backward compatibility.

### Introduction to the JAXB Class Generator

This section provides an introduction to the Java Architecture for XML Binding (JAXB). It discusses the following topics:

- Prerequisites
- Standards and Specifications
- Marshalling and Unmarshalling with JAXB
- Validation with JAXB
- JAXB Customization

### Prerequisites

This chapter assumes that you already have some familiarity with the following topics:

- **Java Architecture for XML Binding (JAXB).** If you require a more thorough introduction to JAXB than is possible in this chapter, consult the XML resources listed in "Related Documents" on page xxxvii of the preface.

- **XML Schema language.** Refer to Chapter 7, "Using the Schema Processor for Java" for an overview and links to suggested reading.

### Standards and Specifications

The Oracle JAXB processor implements JSR-31 "The Java Architecture for XML Binding (JAXB)", Version 1.0, which is a recommendation of the JCP (Java Community Process).
The Oracle Database XDK implementation of the JAXB 1.0 specification does not support the following optional features:

- Javadoc generation
- Fail Fast validation
- External customization file
- XML Schema concepts described in section E.2 of the specification

**JSR** is a Java Specification Request of the JCP. You can find a description of the JSR at the following URL:

http://jcp.org/en/jsr/overview

**See Also:** Chapter 31, "XDK Standards" for a summary of the standards supported by the XDK

### JAXB Class Generator Features

The JAXB class generator for Java generates the interfaces and the implementation classes corresponding to an XML Schema. Its principal advantage to Java developers is automation of the mapping between XML documents and Java code, which enables programs to use generated code to read, manipulate, and re-create XML data. The Java classes, which can be extended, give the developer access to the XML data without knowledge of the underlying XML data structure.

In short, the Oracle JAXB class generator provides the following advantages for XML application development in Java:

- **Speed**
  Because the schema-to-code conversion is automated, you can rapidly generate Java code from an input XML schema.

- **Ease of use**
  You can call generated `get` and `set` methods rather than code your own from scratch.

- **Automated data conversion**
  You can automate the conversion of XML document data into Java datatypes.

- **Customization**
  JAXB provides a flexible framework that enables you to customize the binding of XML elements and attributes.

### Marshalling and Unmarshalling with JAXB

JAXB is an API and set of tools that maps XML data to Java objects. JAXB simplifies access to an XML document from a Java program by presenting the XML document to the program in a Java format.

You can use the JAXB API and tools to perform the following basic tasks:

1. **Generate and compile JAXB classes from an XML schema with the `orajaxb` command-line utility.**

   To use the JAXB class generator to generate Java classes you must provide it with an XML schema. DTDs are not supported by JAXB. As explained in "Converting DTDs to XML Schemas" on page 7-23, however, you can use the `DTD2Schema` tool.
program to convert a DTD to an XML schema. Afterwards, you can use the JAXB class generator to generate classes from the schema.

The JAXB compiler generates Java classes that map to constraints in the source XML schema. The classes implements `get` and `set` methods that you can use to obtain and specify data for each type of element and attribute in the schema.

2. Process XML documents by instantiating the generated classes in a Java program. Specifically, you can write a program that uses the JAXB binding framework to perform the following tasks:

   a. Unmarshal the XML documents.

      As explained in the JAXB specification, **unmarshalling** is defined as moving data from an XML document to the Java-generated objects.

   b. Validate the XML documents.

      You can validate before or during the unmarshalling of the contents into the content tree. You can also validate on demand by calling the validation API on the Java object. Refer to "Validation with JAXB" on page 8-3.

   c. Modify Java content objects.

      The content tree of data objects represents the structure and content of the source XML documents. You can use the `set` methods defined for a class to modify the content of elements and attributes.

   d. Marshal Java content objects back to XML.

      In contrast to unmarshalling, **marshalling** is creating an XML document from Java objects by traversing a content tree of instances of Java classes. You can serialize the data to a DOM tree, SAX content handler, transformation result, or output stream.

**Validation with JAXB**

A Java content tree is considered valid with respect to an XML schema when marshalling the tree generates a valid XML document.

JAXB applications can perform validation in the following circumstances:

- Unmarshalling-time validation that notifies the application of errors and warnings during unmarshalling. If unmarshalling includes validation that is error-free, then the input XML document and the Java content tree are valid.

- On-demand validation of a Java content tree initiated by the application.

- Fail-fast validation that gives immediate results while updating the Java content tree with `set` and `get` methods. As specified in "Standards and Specifications" on page 8-1, fail-fast validation is an optional feature in the JAXB 1.0 specification that is not supported in the XDK implementation of the JAXB class generator.

JAXB applications must be able to marshal a valid Java content tree, but they are not required to ensure that the Java content tree is valid before calling one of the marshalling APIs. The marshalling process does not itself validate the content tree. Programs are merely required to throw a `javax.xml.bind.MarshalException` when marshalling fails due to invalid content.
JAXB Customization

The declared element and type names in an XML schema do not always provide the most useful Java class names. You can override the default JAXB bindings by using custom binding declarations, which are described in the JAXB specification. These declarations enable you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java-specific refinements such as class and package name mappings.

You can annotate the schema to perform the following customizations:

- Bind XML names to user-defined Java class names
- Name the package, derived classes, and methods
- Choose which elements to bind to which classes
- Decide how to bind each attribute and element declaration to a property in the appropriate content class
- Choose the type of each attribute-value or content specification

Several of the demos programs listed in Table 8–2 illustrate JAXB customizations.

See Also:

- "Customizing a Class Name in a Top-Level Element" on page 8-13 for a detailed explanation of a customization demo

Using the JAXB Class Generator: Overview

This section contains the following topics:

- Using the JAXB Processor: Basic Process
- Running the XML Schema Processor Demo Programs
- Using the JAXB Class Generator Command-Line Utility

Using the JAXB Processor: Basic Process

The XDK JAXB API exposes the following packages:

- javax.xml.bind, which provides a runtime binding framework for client applications including unmarshalling, marshalling, and validation
- javax.xml.bind.util, which provides useful client utility classes

The most important classes and interfaces in the javax.xml.bind package are described in Table 8–1. These form the core of most JAXB applications.
Table 8–1  javax.xml.bind Classes and Interfaces

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Description</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAXBContext class</td>
<td>Provides an abstraction for managing the XML/Java binding information necessary to implement the JAXB binding framework operations: unmarshal, marshal, and validate. A client application obtains new instances of this class by invoking the newInstance() method.</td>
<td>The principal methods are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getInstance() creates a JAXB content class. Supply this method the name of the package containing the generated classes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ createMarshaller() creates a marshaller that you can use to convert a content tree to XML.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ createUnmarshaller() creates an unmarshaller that you can use to convert XML to a content tree.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ createValidator() creates a Validator object that can validate a java content tree against its source schema.</td>
</tr>
<tr>
<td>Marshaller interface</td>
<td>Governs the process of serializing Java content trees into XML data.</td>
<td>The principal methods are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getEventHandler() returns the current or default event handler.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getProperty() obtains the property in the underlying implementation of marshaller.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ marshal() marshals the content tree into a DOM, SAX2 events, output stream, transformation result, or Writer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ setEventHandler() creates a Validator object that validates a java content tree against its source schema.</td>
</tr>
<tr>
<td>Unmarshaller interface</td>
<td>Governs the process of deserializing XML data into newly created Java content trees, optionally validating the XML data as it is unmarshalled.</td>
<td>The principal methods are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getEventHandler() returns the current or default event handler.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getUnmarshallerHandler() returns an unmarshaller object usable as a component in an XML pipeline.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ isValidating() indicates whether the unmarshaller is set to validate mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ setEventHandler() allows an application to register a ValidationEventHandler.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ setValidating() specifies whether the unmarshaller should validate during unmarshal operations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ marshal() unmarshals XML data from the specified file, URL, input stream, input source, SAX, or DOM.</td>
</tr>
<tr>
<td>Validator interface</td>
<td>Controls the validation of content trees during runtime. Specifically, this interface controls on-demand validation, which enables clients to receive data about validation errors and warnings detected in the Java content tree.</td>
<td>The principal methods are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ getEventHandler() returns the current or default event handler.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ setEventHandler() allows an application to register a ValidationEventHandler.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ validate() validates Java content trees on-demand at runtime. This method can validate any arbitrary subtree of the Java content tree.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ validateRoot() validates the Java content tree rooted at rootObj. You can use this method to validate an entire Java content tree.</td>
</tr>
</tbody>
</table>

Figure 8–1 depicts the process flow of a framework that uses the JAXB class generator.
The basic stages of the process illustrated in Figure 8–1 are as follows:

1. The XML parser parses the XML schema and sends the parsed data to the JAXB class generator.

2. The class generator creates Java classes and interfaces based on the input XML schema.

   By default, one XML element or type declaration generates one interface and one class. For example, if the schema defines an element named `<anElement>`, then by default the JAXB class generator generates a source file named `AnElement.java` and another named `AnElementImpl.java`. You can use customize binding declarations to override the default binding of XML Schema components to Java representations.

3. The Java compiler compiles the `.java` source files into class files. All of the generated classes, source files, and application code must be compiled.

4. Your Java application uses the compiled classes and the binding framework to perform the following types of tasks:
   - Create a JAXB context. You use this context to create the marshaller and unmarshaller.
   - Build object trees representing XML data that is valid against the XML schema. You can perform this task by either unmarshalling the data from an XML document that conforms to the schema or instantiating the classes.
   - Access and modify the data.
   - Optionally validate the modifications to the data relative to the constraints expressed in the XML schema.
   - Marshal the data to new XML documents.

See Also:

- [http://java.sun.com/xml/jaxb/faq.html](http://java.sun.com/xml/jaxb/faq.html) for more information on JAXB
- [Oracle Database XML Java API Reference](http://www.oracle.com/technetwork/database/orm/jaxb-ref-107586.html) for details of the JAXB API
- "Processing XML with the JAXB Class Generator" on page 8-9 for detailed explanations of JAXB processing
Running the XML Schema Processor Demo Programs

Demo programs for the JAXB class generator for Java are included in $ORACLE_HOME/xdk/demo/java/jaxb. Specifically, the XDK includes the JAXB demos listed in Table 8–2.

Table 8–2  JAXB Class Generator Demos

<table>
<thead>
<tr>
<th>Program</th>
<th>Subdirectory within Oracle Home</th>
<th>Demonstrates . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleApp1.java</td>
<td>/xdk/demo/java/jaxb/Sample1</td>
<td>The binding of top-level element and complexType definitions in the sample1.xsd schema to Java classes.</td>
</tr>
<tr>
<td>SampleApp2.java</td>
<td>/xdk/demo/java/jaxb/Sample2</td>
<td>The binding of a top-level element with an inline simpleType definition in the sample2.xsd schema.</td>
</tr>
<tr>
<td>SampleApp3.java</td>
<td>/xdk/demo/java/jaxb/Sample3</td>
<td>The binding of a top-level complexType element that is derived by extension from another top-level complexType definition. Refer to “Binding Complex Types” on page 8-10 for a detailed explanation of this program.</td>
</tr>
<tr>
<td>SampleApp4.java</td>
<td>/xdk/demo/java/jaxb/Sample4</td>
<td>The binding of a content model within a complexType that refers to a top-level named group.</td>
</tr>
<tr>
<td>SampleApp5.java</td>
<td>/xdk/demo/java/jaxb/Sample5</td>
<td>The binding of &lt;choice&gt; with maxOccurs unbounded within a complexType.</td>
</tr>
<tr>
<td>SampleApp6.java</td>
<td>/xdk/demo/java/jaxb/Sample6</td>
<td>The binding of atomic datatypes.</td>
</tr>
<tr>
<td>SampleApp7.java</td>
<td>/xdk/demo/java/jaxb/Sample7</td>
<td>The binding a complexType definition in which mixed=&quot;true&quot;.</td>
</tr>
<tr>
<td>SampleApp8.java</td>
<td>/xdk/demo/java/jaxb/Sample8</td>
<td>The binding of elements and types declared in two different namespaces.</td>
</tr>
<tr>
<td>SampleApp9.java</td>
<td>/xdk/demo/java/jaxb/Sample9</td>
<td>The customization of a Java package name.</td>
</tr>
<tr>
<td>SampleApp10.java</td>
<td>/xdk/demo/java/jaxb/Sample10</td>
<td>The customization of class name in a top-level element. Refer to “Customizing a Class Name in a Top-Level Element” on page 8-13 for a detailed explanation of this program.</td>
</tr>
<tr>
<td>SampleApp11.java</td>
<td>/xdk/demo/java/jaxb/Sample11</td>
<td>The customization of class name of a local element occurring in a repeating model group declared inside a complexType element.</td>
</tr>
<tr>
<td>SampleApp12.java</td>
<td>/xdk/demo/java/jaxb/Sample12</td>
<td>The customization of the attribute name.</td>
</tr>
<tr>
<td>SampleApp13.java</td>
<td>/xdk/demo/java/jaxb/Sample13</td>
<td>The javaType customization specified on a global simpleType. The javaType customization specifies the parse and print method declared on a user-defined class.</td>
</tr>
<tr>
<td>SampleApp14.java</td>
<td>/xdk/demo/java/jaxb/Sample14</td>
<td>The customization of the typesafe enum class name.</td>
</tr>
</tbody>
</table>

You can find documentation that describes how to compile and run the sample programs in the README in the same directory. The basic steps are as follows:

1. Change into the $ORACLE_HOME/xdk/demo/java/jaxb directory (UNIX) or %ORACLE_HOME%\xdk\demo\java\jaxb directory (Windows).

2. Make sure that your environment variables are set as described in "Setting Up the Java XDK Environment" on page 3-5.

3. Run make (UNIX) or Make.bat (Windows) at the system prompt. The make utility performs the following sequential actions for each sample subdirectory:

   a. Runs the orajaxb utility to generate Java class files based on an input XML schema. For most of the demos, the output classesfiles are written to the generated subdirectory. For example, the make file performs the following commands for the sample1.xsd schema in the Sample1 subdirectory:

      ```
      cd ./Sample1; $(JAVA_HOME)/bin/java -classpath "${MAKE_CLASSPATH}" \ 
      oracle.xml.jaxb.orajaxb -schema sample1.xsd -targetPkg generated; echo;
      ```
b. Runs the `javac` utility to compile the Java classes. For example, the `make` utility performs the following commands for the Java class files in the `Sample1/generated/` subdirectory:

```bash
cd ./Sample1/generated; $(JAVA_HOME)/bin/javac -classpath "$(MAKE_CLASSPATH)" *.java
```

c. Runs the `javac` utility to compile a sample Java application that uses the classes compiled in the preceding step. For example, the `make` utility compiles the `SampleApp1.java` program:

```bash
cd ./Sample1; $(JAVA_HOME)/bin/javac -classpath "$(MAKE_CLASSPATH)" SampleApp1.java
```

d. Runs the sample Java application and writes the results to a log file. For example, the `make` utility executes the `SampleApp1` class and writes the output to `sample1.out`:

```bash
cd ./Sample1; $(JAVA_HOME)/bin/java -classpath "$(MAKE_CLASSPATH)" SampleApp1 > sample1.out
```

**Using the JAXB Class Generator Command-Line Utility**

The XDK includes `orajaxb`, which is a command-line Java interface that generates Java classes from input XML schemas. The `$ORACLE_HOME/bin/orajaxb` and `%ORACLE_HOME%\bin\orajaxb.bat` shell scripts execute the `oracle.xml.jaxb.orajaxb` class. To use `orajaxb` ensure that your CLASSPATH is set as described in "Setting Up the Java XDK Environment" on page 3-5.

Table 8–3 lists the `orajaxb` command-line options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-help</td>
<td>Prints the help message.</td>
</tr>
<tr>
<td>-version</td>
<td>Prints the release version.</td>
</tr>
<tr>
<td>-outputdir OutputDir</td>
<td>Specifies the directory in which to generate the Java source files. If the schema has a namespace, then the program generates the java code in the package (corresponding to the namespace) referenced from the outputDir. By default, the current directory is the outputDir.</td>
</tr>
<tr>
<td>-schema SchemaFile</td>
<td>Specifies the input XML schema.</td>
</tr>
<tr>
<td>-targetPkg targetPkg</td>
<td>Specifies the target package name. This option overrides any binding customization for package name as well as the default package name algorithm defined in the JAXB Specification.</td>
</tr>
<tr>
<td>-interface</td>
<td>Generates the interfaces only.</td>
</tr>
<tr>
<td>-verbose</td>
<td>Lists the generated classes and interfaces.</td>
</tr>
<tr>
<td>-defaultCus fileName</td>
<td>Generates the default customization file.</td>
</tr>
<tr>
<td>-extension</td>
<td>Allows vendor specific extensions and does not strictly follow the compatibility rules specified in Appendix E.2 of the JAXB 1.0 specification. When specified, the program ignores JAXB 1.0 unsupported features such as notations, substitution groups, and any attributes.</td>
</tr>
</tbody>
</table>
Using the JAXB Class Generator Command-Line Utility: Example

To test orajaxb, change into the $ORACLE_HOME/xdk/demo/java/jaxb/Sample1 directory. If you have run make, then the directory should contain the following files:

SampleApp1.class
SampleApp1.java
generated/
sample1.out
sample1.xml
sample1.xsd

The sample.xsd file is the XML schema associated with sample1.xml. The generated/ subdirectory contains the classes generated from the input schema. You can test orajaxb by deleting the contents of generated/ and regenerating the classes as follows:

rm generated/*
orajaxb -schema sample1.xsd -targetPkg generated -verbose

The terminal should display the following output:

generated/ CType.java
generated/AComplexType.java
generated/AnElement.java
generated/RElemOfCTypeInSameNs.java
generated/RType.java
generated/RElemOfSTypeInSameNs.java

generated/ CTypeImpl.java
generated/AComplexTypeImpl.java
generated/AnElementImpl.java
generated/RElemOfCTypeInSameNsImpl.java
generated/RTypeImpl.java
generated/RElemOfSTypeInSameNsImpl.java
generated/ObjectFactory.java

JAXB Features Not Supported in the XDK

The Oracle Database XDK implementation of the JAXB specification does not support the following features:

- Javadoc generation
- XML Schema component "any" and substitution groups

See Also: http://java.sun.com/xml/downloads/jaxb.html for the JAXB specification

Processing XML with the JAXB Class Generator

This section contains the following topics:

- Binding Complex Types
- Customizing a Class Name in a Top-Level Element
Binding Complex Types

The Sample3.java program illustrates how to bind a complex type definition to a Java content interface. One complex type defined in the XML schema is derived by extension from another complex type.

Defining the Schema

Example 8–1 illustrates the XML data document that provides the input to the sample application. The sample3.xml document describes the address of an employee.

Example 8–1  sample3.xml

```xml
<?xml version="1.0"?>
<myAddress xmlns = "http://www.oracle.com/sample3/"
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation = "http://www.oracle.com/sample3 sample3.xsd">
  <name>James Bond</name>
  <doorNumber>420</doorNumber>
  <street>Oracle parkway</street>
  <city>Redwood shores</city>
  <state>CA</state>
  <zip>94065</zip>
  <country>United States</country>
</myAddress>
```

The XML schema shown in Example 8–2 defines the structure that you use to validate sample3.xml. The schema defines two complex types and one element, which are defined as follows:

- The first complex type, which is named Address, is a sequence of elements. Each element in the sequence describes one part of the address: name, door number, and so forth.

- The second complex type, which is named USAddress, uses the `<extension base="exp:Address">` element to extend Address by adding U.S.-specific elements to the Address sequence: state, zip, and so forth. The exp prefix specifies the http://www.oracle.com/sample3/ namespace.

- The element is named myAddress and is of type exp:USAddress. The exp prefix specifies the http://www.oracle.com/sample3/ namespace. In sample3.xml, the myAddress top-level element, which is in namespace http://www.oracle.com/sample3/, conforms to the schema definition.

Example 8–2  sample3.xsd

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

<!-- Binding a complex type definition to java content interface
The complex type definition is derived by extension -->

<schema xmlns = "http://www.w3.org/2001/XMLSchema"
xmlns:exp="http://www.oracle.com/sample3/"
targetNamespace="http://www.oracle.com/sample3/
  elementFormDefault="qualified">

  <complexType name="Address">
    <sequence>
      <element name="name" type="string"/>
      <element name="doorNumber" type="short"/>
    </sequence>
  </complexType>

```

8-10 Oracle XML Developer’s Kit Programmer’s Guide
<element name="street" type="string"/>
<element name="city" type="string"/>
</sequence>
</complexType>

<complexType name="USAddress">
<complexContent>
<extension base="exp:Address">
<sequence>
<element name="state" type="string"/>
<element name="zip" type="integer"/>
<element name="country" type="string"/>
</sequence>
</extension>
</complexContent>
</complexType>

<element name="myAddress" type="exp:USAddress"/>
</schema>

Generating and Compiling the Java Classes
If you have an XML document and corresponding XML schema, then the next stage of processing is to generate the Java classes from the XML schema. You can use the JAXB command-line interface described in "Using the JAXB Class Generator Command-Line Utility" on page 8-8 to perform this task.

Assuming that your environment is set up as described in "Setting Up the Java XDK Environment" on page 3-5, you can create the source files in the generated package as follows:

cd $ORACLE_HOME/xdk/demo/java/jaxb/Sample3
orajaxb -schema sample1.xsd -targetPkg generated

The preceding orajaxb command should create the following source files in the ./generated/ subdirectory:
Address.java
AddressImpl.java
MyAddress.java
MyAddressImpl.java
ObjectFactory.java
USAddress.java
USAddressImpl.java

The complex types Address and USAddress each has two associated source files, as does the element MyAddress. The source file named after the element contains the interface; the file with the suffix Impl contains the class that implements the interface. For example, Address.java contains the interface Address, whereas AddressImpl.java contains the class that implements Address.

The content of the Address.java source file is shown in Example 8–3.

Example 8–3 Address.java
package generated;
public interface Address
{
   public void setName(java.lang.String n);
   public java.lang.String getName();
}
public void setDoorNumber(short d);  
public short getDoorNumber();  
public void setStreet(java.lang.String s);  
public java.lang.String getStreet();  
public void setCity(java.lang.String c);  
public java.lang.String getCity();
}

The Address complex type defined a sequence of elements: name, doorNumber, street, and city. Consequently, the Address interface includes a get and set method signature for each of the four elements. For example, the interface includes getName() for retrieving data in the <name> element and setName() for modifying data in this element.

You can compile the Java source files with javac as follows:

```
cd $ORACLE_HOME/xdk/demo/java/jaxb/Sample3/generated  
javac *.java
```

### Processing the XML Data

Sample3.java shows how you can process the sample3.xml document by using the Java class files that you generated in "Generating and Compiling the Java Classes" on page 8-11. The sample program unmarshals the XML data document, marshals it, and uses the generated classes to print and modify the address data.

The Sample3.java program processes the data as follows:

1. Create strings for the XML data document file name and the name of the directory that contains the generated classes. This name is the package name. For example:

   ```java
   String fileName = "sample3.xml";
   String instancePath = "generated";
   ```

2. Instantiate a JAXB context by invoking JAXBContext.newInstance(). A client application obtains a new instance of this class by initializing it with a context path. The path contains a list of Java package names that contain the interfaces available to the marshaller. The following statement illustrates this technique:

   ```java
   JAXBContext jc = JAXBContext.newInstance(instancePath);
   ```

3. Instantiate the unmarshaller. The Unmarshaller class governs the process of deserializing XML data into newly created objects, optionally validating the XML data as it is unmarshalled. The following statement illustrates this technique:

   ```java
   Unmarshaller u = jc.createUnmarshaller();
   ```

4. Unmarshal the XML document. Invoke the Unmarshaller.unmarshal() method to deserialize the sample3.xml document and return the content trees as an Object. You can create a URL from the XML filename by invoking the fileToUrl() helper method. The following statement illustrates this technique:

   ```java
   Object obj = u.unmarshal(fileToURL(fileName));
   ```

5. Instantiate a marshaller. The Marshaller class governs the process of serializing Java content trees back into XML data. The following statement illustrates this technique:

   ```java
   Marshaller m = jc.createMarshaller();
   ```

6. Marshal the content tree. Invoke the Marshaller.marshal() method to marshal the content tree Object returned by the unmarshaller. You can serialize the data to
Processing XML with the JAXB Class Generator

a DOM tree, SAX content handler, transformation result, or output stream. The following statement serializes the XML data, including markup, as an output stream:

```java
m.marshal(obj, System.out);
```

By default, the marshaller uses UTF-8 encoding when writing XML data to an output stream.

7. Print the contents of the XML document. The program implements a `process()` method that accepts the content tree and marshaller as parameters.

The first stage of processing prints the data in the XML document without the XML markup. The method casts the `Object` generated by the marshaller into type `MyAddress`. It proceeds to invoke a series of methods whose method names are constructed by prefixing `get` to the name of an XML element. For example, to obtain the data in the `<city>` element in Example 8–1, the program invokes `getCity()`. The following code fragment illustrates this technique:

```java
public static void process(Object obj, Marshaller m) throws Throwable {
    generated.MyAddress elem = (generated.MyAddress)obj;
    System.out.println();
    System.out.println(" My address is: ");
    System.out.println(" name: " + elem.getName() + "\n"
        + " doorNumber " + elem.getDoorNumber() + "\n"
        + " street: " + elem.getStreet() + "\n"
        + " city: " + elem.getCity() + "\n"
        + " state: " + elem.getState() + "\n"
        + " zip: " + elem.getZip() + "\n"
        + " country: " + elem.getCountry() + "\n"
        + "\n" );
    ...
}
```

8. Change the XML data and print it. The `process()` method continues by invoking set methods that are analogous to the preceding get methods. The name of each set method is constructed by prefixing `set` to the name of an XML element. For example, `setCountry()` changes the value in the `<country>` element. The following statements illustrate this technique:

```java
short num = 550;
elem.setDoorNumber(num);
elem.setCountry("India");
num = 10100;
elem.setZip(new java.math.BigInteger("100100"));
elem.setCity("Noida");
elem.setState("Delhi");
```

After changing the data, the program prints the data by invoking the same get methods as in the previous step.

**Customizing a Class Name in a Top-Level Element**

The `Sample10.java` program illustrates one form of JAXB customization. The program shows you can change the name of a class that corresponds to an element in the input XML schema.
Defining the Schema

Example 8–4 shows the XML data document that provides the input to the sample application. The `sample10.xml` document describes a business.

**Example 8–4  sample10.xml**

```xml
<?xml version="1.0"?>
<business xmlns="http://jaxbcustomized/sample10/">
    <title>Software Development</title>
    <owner>Larry Peterson</owner>
    <id>45123</id>
</business>
```

Example 8–5 shows the XML schema that defines the structure of `sample10.xml`. The schema defines one complex type and one element as follows:

- The complex type, which is named `businessType`, is a sequence of elements. Each element in the sequence describes a part of the business: title, owner, and id.
- The element, which is named `business`, is of type `biz:businessType`. The `biz` prefix specifies the `http://jaxbcustomized/sample10/` namespace. In `sample10.xml`, the `business` top-level element, which is in namespace `http://jaxbcustomized/sample10/`, conforms to the schema definition.

**Example 8–5  sample10.xsd**

```xml
<?xml version="1.0"?>
<!-- Customization of class name in top level element -->

<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://jaxbcustomized/sample10/"
    xmlns:biz="http://jaxbcustomized/sample10/"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    jaxb:version="1.0"
    elementFormDefault="qualified">

    <element name="business" type="biz:businessType">
        <annotation>
            <appinfo>
                <jaxb:class name="myBusiness"/>
            </appinfo>
        </annotation>
    </element>

    <complexType name="businessType">
        <sequence>
            <element name="title" type="string"/>
            <element name="owner" type="string"/>
            <element name="id" type="integer"/>
        </sequence>
    </complexType>

</schema>
```

Customizing the Schema Binding  The schema shown in Example 8–5 customizes the binding of the `business` element by means of an inline binding declaration. The general form for inline customizations is the following:

```xml
<xs:annotation>
Example 8–5 uses the `<class>` binding declaration to bind a schema element to a Java class name. You can use the declaration to customize the name for an interface or the class that implements an interface. The JAXB class generator supports the following syntax for `<class>` customizations:

```
<class [ name = "className"] >
```

The `name` attribute specifies the name of the derived Java interface. Example 8–5 contains the following customization:

```
<jaxb:class name="myBusiness"/>
```

Thus, the schema binds the `business` element to the interface `myBusiness` rather than to the interface `business`, which is the default.

### Generating and Compiling the Java Classes

After you have an XML document and corresponding XML schema, the next stage is to generate the Java classes from the XML schema. You can use the JAXB command-line interface to perform this task.

If your environment is set up as described in "Setting Up the Java XDK Environment" on page 3-5, then you can create the source files in the generated package as follows:

```
cd $ORACLE_HOME/xdk/demo/java/jaxb/Sample10
orajAXB -schema sample10.xsd
```

Because the preceding command does not specify a target package, the package name is constructed from the target namespace of the schema, which is `http://jaxbcustomized/sample10/`. Consequently, the utility generates the following source files in the `.jaxbcustomized/sample10/` subdirectory:

- `BusinessType.java`
- `BusinessTypeImpl.java`
- `MyBusiness.java`
- `MyBusinessImpl.java`
- `ObjectFactory.java`

Note that the complex type `businessType` has two source files, `BusinessType.java` and `BusinessTypeImpl.java`. Because of the JAXB customization, the `business` element is bound to interface `MyBusiness` and implementing class `MyBusinessImpl`.

The content of the `BusinessType.java` source file is shown in Example 8–6.

### Example 8–6  BusinessType.java

```java
package jaxbcustomized.sample10;

public interface BusinessType
{
    public void setTitle(java.lang.String t);
    public java.lang.String getTitle();
}
```
The `BusinessType` complex type defined a sequence of elements: `title`, `owner`, and `id`. Consequently, the Address interface includes a get and set method signature for each of the elements. For example, the interface includes `getTitle()` for retrieving data in the `<title>` element and `setTitle()` for modifying data in this element.

You can compile the Java source files with `javac` as follows:

```
cd $ORACLE_HOME/xdk/demo/java/jaxb/Sample10/jaxbcustomized/sample10
javac *.java
```

## Processing the XML Data

The `Sample10.java` source file shows how you can process the data in the `sample10.xml` document by using the class files that you generated in "Generating and Compiling the Java Classes" on page 8-15. The sample program unmarshals the XML document, prints its content, and marshals the XML to standard output.

The `Sample10.java` program processes the XML data as follows:

1. Create strings for the XML data document file name and the name of the directory that contains the generated classes. This name is the package name. For example:

   ```java
   String fileName = "sample10.xml";
   String instancePath = "jaxbcustomized.sample10";
   ```

2. Instantiate a JAXB context by invoking the `JAXBContext.newInstance()` method. The following statement illustrates this technique:

   ```java
   JAXBContext jc = JAXBContext.newInstance(instancePath);
   ```

3. Create the unmarshaller. The following statement illustrates this technique:

   ```java
   Unmarshaller u = jc.createUnmarshaller();
   ```

4. Unmarshal the XML document. The program unmarshals the document twice: it first returns an `Object` and then uses a cast to return a `MyBusiness` object. The following statement illustrates this technique:

   ```java
   Object obj = u.unmarshal(fileToURL(fileName));
   jaxbcustomized.sample10.MyBusiness bus =
   (jaxbcustomized.sample10.MyBusiness) u.unmarshal(fileToURL(fileName));
   ```

5. Print the contents of the XML document. The program invokes the `get` methods on the `MyBusiness` object. The following code fragment illustrates this technique:

   ```java
   System.out.println("My business details are: ");
   System.out.println("    title: "+ bus.getTitle());
   System.out.println("    owner: "+ bus.getOwner());
   System.out.println("    id: "+ bus.getId().toString());
   System.out.println();
   ```

6. Create a marshaller. The following statement illustrates this technique:

   ```java
   Marshaller m = jc.createMarshaller();
   ```

7. Configure the marshaller. You can invoke `setProperty()` to configure various properties the marshaller. The `JAXB_FORMATTED_OUTPUT` constant specifies that the
marshaller should format the resulting XML data with line breaks and indentation. The following statements illustrate this technique:

```
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, new Boolean(true));
```

8. Marshal the content tree. The following statement serializes the XML data, including markup, as an output stream:

```
m.marshal(bus, System.out);
```

By default, the marshaller uses UTF-8 encoding when writing XML data to an output stream.
Using the XML Pipeline Processor for Java

This chapter contains these topics:

- Introduction to the XML Pipeline Processor
- Using the XML Pipeline Processor: Overview
- Processing XML in a Pipeline

Introduction to the XML Pipeline Processor

This section contains the following topics:

- Prerequisites
- Standards and Specifications
- Multistage XML Processing
- Customized Pipeline Processes

Prerequisites

This chapter assumes that you are familiar with the following topics:

- **XML Pipeline Definition Language.** This XML vocabulary enables you to describe the processing relations between XML resources. If you require a more thorough introduction to the Pipeline Definition Language, consult the XML resources listed in "Related Documents" on page xxxvii of the preface.

- **Document Object Model (DOM).** DOM is an in-memory tree representation of the structure of an XML document.

- **Simple API for XML (SAX).** SAX is a standard for event-based XML parsing.

- **XML Schema language.** Refer to Chapter 7, "Using the Schema Processor for Java" for an overview and links to suggested reading.

Standards and Specifications

The Oracle XML Pipeline processor is based on the W3C XML Pipeline Definition Language Version 1.0 Note. The W3C Note defines an XML vocabulary rather than an API. You can find the Pipeline specification at the following URL:

http://www.w3.org/TR/xml-pipeline/

"Pipeline Definition Language Standard for the XDK for Java" on page 31-5 describes the differences between the Oracle XDK implementation of the Oracle XML Pipeline processor and the W3C Note.
See Also: Table 31–1, "Summary of XML Standards Supported by the XDK"

Multistage XML Processing

The Oracle XML Pipeline processor is built on the XML Pipeline Definition Language. The processor can take an input XML pipeline document and execute pipeline processes according to derived dependencies. A pipeline document, which is written in XML, specifies the processes to be executed in a declarative manner. You can associate Java classes with processes by using the <processdef/> element in the pipeline document.

Use the Pipeline processor for mutistage processing, which occurs when you process XML components sequentially or in parallel. The output of one stage of processing can become the input of another stage of processing. You can write a pipeline document that defines the inputs and outputs of the processes. Figure 9–1 illustrates a possible pipeline sequence.

![Pipeline Processing Diagram](Figure 9–1]

In addition to the XML Pipeline processor itself, the XDK provides an API for processes that you can pipe together in a pipeline document. Table 9–2 summarizes the classes provided in the oracle.xml.pipeline.processes package.

The typical stages of processing XML in a pipeline are as follows:

2.  Validate the input XML documents.
3.  Serialize or transform the input documents. Note that the Pipeline processor does not enable you to connect the SAX parser to the XSLT processor, which requires a DOM.

In multistage processing, SAX is ideal for filtering and searching large XML documents. You should use DOM when you need to change XML content or require efficient dynamic access to the content.

See Also: "Processing XML in a Pipeline" on page 9-9 to learn how to write a pipeline document that provides the input for a pipeline application

Customized Pipeline Processes

The oracle.xml.pipeline.controller.Process class is the base class for all pipeline process definitions. The classes in the oracle.xml.pipeline.processes package
extend this base class. To create a customized pipeline process, you need to create a class that extends the `Process` class.

At the minimum, every custom process should override the do-nothing `initialize()` and `execute()` methods of the `Process` class. If the customized process accepts SAX events as input, then it should override the `SAXContentHandler()` method to return the appropriate `ContentHandler` that handles incoming SAX events. It should also override the `SAXErrorHandler()` method to return the appropriate `ErrorHandler`. Table 9–1 provides further descriptions of the preceding methods.

**Table 9–1  Methods in the oracle.xml.pipeline.controller.Process Class**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize()</td>
<td>Initializes the process before execution.</td>
</tr>
<tr>
<td></td>
<td>Call <code>getInput()</code> to fetch a specific input object associated with the process element and call <code>supportType()</code> to indicate the types of input supported. Analogously, call <code>getOutput()</code> and <code>supportType()</code> for output.</td>
</tr>
<tr>
<td>execute()</td>
<td>Executes the process.</td>
</tr>
<tr>
<td></td>
<td>Call <code>getInParaValue()</code>, <code>getInput()</code>, or <code>getInputSource()</code> to fetch the inputs to the process. If a custom process outputs SAX events, then it should call the <code>getSAXContentHandler()</code> and <code>getSAXErrorHandler()</code> methods in <code>execute()</code> to get the SAX handlers of the following processes in the pipeline.</td>
</tr>
<tr>
<td></td>
<td>Call <code>setOutputResult()</code>, <code>getOutputStream()</code>, <code>getOutputWriter()</code> or <code>setOutParam()</code> to set the outputs or outparams generated by this process.</td>
</tr>
<tr>
<td></td>
<td>Call <code>getErrorSource()</code>, <code>getErrorStream()</code>, or <code>getErrorDocument()</code> to access the pipeline error element associated with this process element. If an exception occurs during <code>execute()</code>, call <code>error()</code> or <code>info()</code> to propagate it to the <code>PipelineErrorHandler</code>.</td>
</tr>
<tr>
<td>SAXContentHandler()</td>
<td>Returns the SAX ContentHandler.</td>
</tr>
<tr>
<td></td>
<td>If dependencies from other processes are not available at this time, then return <code>null</code>. When these dependencies are available the method will be executed till the end.</td>
</tr>
<tr>
<td>SAXErrorHandler()</td>
<td>Returns the SAX ErrorHandler.</td>
</tr>
<tr>
<td></td>
<td>If you do not override this method, then the JAXB processor uses the default error handler implemented by this class to handle SAX errors.</td>
</tr>
</tbody>
</table>

**See Also:** *Oracle Database XML Java API Reference* to learn about the `oracle.xml.pipeline.processes` package

### Using the XML Pipeline Processor: Overview

This section contains the following topics:

- Using the XML Pipeline Processor: Basic Process
- Running the XML Pipeline Processor Demo Programs
- Using the XML Pipeline Processor Command-Line Utility

### Using the XML Pipeline Processor: Basic Process

The XML Pipeline processor is accessible through the following packages:

- `oracle.xml.pipeline.controller`, which provides an XML Pipeline controller that executes XML processes in a pipeline based on dependencies.
- `oracle.xml.pipeline.processes`, which provides wrapper classes for XML processes that can be executed by the XML Pipeline controller. The
oracle.xml.pipeline.processes package contains the classes that you can use to design a pipeline application framework. Each class extends the oracle.xml.pipeline.controller.Process class.

Table 9–2 lists the components in the package. You can connect these components and processes through a combination of the XML Pipeline processor and a pipeline document.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompressReaderProcess</td>
<td>Receives compressed XML and outputs parsed XML.</td>
</tr>
<tr>
<td>CompressWriterProcess</td>
<td>Receives XML parsed with DOM or SAX and outputs compressed XML.</td>
</tr>
<tr>
<td>DOMParserProcess</td>
<td>Parses incoming XML and outputs a DOM tree.</td>
</tr>
<tr>
<td>SAXParserProcess</td>
<td>Parses incoming XML and outputs SAX events.</td>
</tr>
<tr>
<td>XPathProcess</td>
<td>Accepts a DOM as input, uses an XPath pattern to select one or more nodes from an XML Document or an XML DocumentFragment, and outputs a Document or DocumentFragment.</td>
</tr>
<tr>
<td>XSDSchemaBuilder</td>
<td>Parses an XML schema and outputs a schema object for validation. This process is built into the XML Pipeline processor and builds schema objects used for validating XML documents.</td>
</tr>
<tr>
<td>XSDValProcess</td>
<td>Validates against a local schema, analyzes the results, and reports errors if necessary.</td>
</tr>
<tr>
<td>XSLProcess</td>
<td>Accepts DOM as input, applies an XSL stylesheet, and outputs the result of the transformation.</td>
</tr>
<tr>
<td>XSLStylesheetProcess</td>
<td>Receives an XSL stylesheet as a stream or DOM and creates an XSLStylesheet object.</td>
</tr>
</tbody>
</table>

Figure 9–2 illustrates how to pass a pipeline document to a Java application that uses the XML Pipeline processor, configure the processor, and execute the pipeline.
The basic steps are as follows:

1. Instantiate a pipeline document, which forms the input to the pipeline execution. Create the object by passing a `FileReader` to the constructor as follows:

   ```java
   PipelineDoc pipe;
   FileReader f;
   pipe = new PipelineDoc((Reader)f, false);
   ```

2. Instantiate a pipeline processor. `PipelineProcessor` is the top-level class that executes the pipeline. Table 9–3 describes some of the available methods.
Using the XML Pipeline Processor: Overview

The following statement instantiates the pipeline processor:

```java
proc = new PipelineProcessor();
```

3. Set the processor to the pipeline document. For example:

```java
proc.setPipelineDoc(pipe);
```

4. Set the execution mode for the processor and perform any other needed configuration. For example, set the mode by passing a constant to `PipelineProcessor.setExecutionMode()`.

```java
proc.setExecutionMode(PipelineConstants.PIPELINE_SEQUENTIAL);
```

5. Instantiate an error handler. The error handler must implement the `PipelineErrorHandler` interface. For example:

```java
errHandler = new PipelineSampleErrHdlr(logname);
```

6. Set the error handler for the processor by invoking `setErrorHandler()`. For example:

```java
proc.setErrorHandler(errHandler);
```

7. Execute the pipeline. For example:

```java
proc.executePipeline();
```

See Also:

- Oracle Database XML Java API Reference to learn about the `oracle.xml.pipeline` subpackages
- "Creating a Pipeline Document" on page 9-9

Running the XML Pipeline Processor Demo Programs

Demo programs for the XML Pipeline processor are included in `$ORACLE_HOME/xdk/demo/java/pipeline`. Table 9–4 describes the XML files and Java source files that you can use to test the utility.
Documentation for how to compile and run the sample programs is located in the README. The basic steps are as follows:
1. Change into the $ORACLE_HOME/xdk/demo/java/pipeline directory (UNIX) or %ORACLE_HOME%/xdk/demo/java/pipeline directory (Windows).

2. Make sure that your environment variables are set as described in "Setting Up the Java XDK Environment" on page 3-5.

3. Run make (UNIX) or Make.bat (Windows) at the system prompt to generate class files for PipelineSample.java and PipelineSampleErrHdler.java and run the demo programs. The programs write output files to the log subdirectory. Alternatively, you can run the demo programs manually by using the following syntax:

   java PipelineSample pipedoc pipelog [ seq | para ]

   The pipedoc option specifies which pipeline document to use. The pipelog option specifies the name of the pipeline log file, which is optional unless you specify seq or para, in which case a filename is required. If you do not specify a log file, then the program generates pipeline.log by default. The seq option processes threads sequentially; para processes in parallel. If you specify neither seq or para, then the default is parallel processing.

4. View the files generated from the pipeline, which are all named with the initial string myresult, and the log files.

Using the XML Pipeline Processor Command-Line Utility

The command-line interface for the XML Pipeline processor is named orapipe. The Pipeline processor is packaged with Oracle database. By default, the Oracle Universal Installer installs the utility on disk in $ORACLE_HOME/bin.

Before running the utility for the first time, make sure that your environment variables are set as described in "Setting Up the Java XDK Environment" on page 3-5. Run orapipe at the operating system command line with the following syntax:

   orapipe options pipedoc

   The pipedoc is the pipeline document, which is required. Table 9-5 describes the available options for the orapipe utility.

<table>
<thead>
<tr>
<th>Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-help</td>
<td>Prints the help message</td>
</tr>
<tr>
<td>-log logfile</td>
<td>Writes errors and messages to the specified log file. The default is pipeline.log.</td>
</tr>
<tr>
<td>-noinfo</td>
<td>Does not log informational items. The default is on.</td>
</tr>
<tr>
<td>-nowarning</td>
<td>Does not log warnings. The default is on.</td>
</tr>
<tr>
<td>-validate</td>
<td>Validates the input pipedoc with the pipeline schema. Validation is turned off by default. If outparam feature is used, then validate fails with the current pipeline schema because this is an additional feature.</td>
</tr>
<tr>
<td>-version</td>
<td>Prints the release version</td>
</tr>
<tr>
<td>-sequential</td>
<td>Executes the pipeline in sequential mode. The default is parallel.</td>
</tr>
<tr>
<td>-force</td>
<td>Executes pipeline even if target is up-to-date. By default no force is specified.</td>
</tr>
</tbody>
</table>
Processing XML in a Pipeline

This section contains the following topics:

- Creating a Pipeline Document
- Writing a Pipeline Processor Application
- Writing a Pipeline Error Handler

Creating a Pipeline Document

To use the Oracle XML Pipeline processor, you must create an XML document according to the rules of the Pipeline Definition Language specified in the W3C Note. The W3C specification defines the XML processing components and the inputs and outputs for these processes. The XML Pipeline processor includes support for the following XDK components:

- XML parser
- XML compressor
- XML Schema validator
- XSLT processor

Example of a Pipeline Document

The XML Pipeline processor executes a sequence of XML processing according to the rules in the pipeline document and returns a result. Example 9–1 shows pipedoc.xml, which is a sample pipeline document included in the demo directory.

Example 9–1  pipedoc.xml

```xml
<pipeline xmlns="http://www.w3.org/2002/02/xml-pipeline"
          xml:base="http://example.org/">

  <param name="target" select="myresult.html"/>

  <processdef name="domparser.p"
              definition="oracle.xml.pipeline.processes.DOMParserProcess"/>
  <processdef name="xslstylesheet.p"
              definition="oracle.xml.pipeline.processes.XSLStylesheetProcess"/>
  <processdef name="xslprocess.p"
              definition="oracle.xml.pipeline.processes.XSLProcess"/>

  <process id="p2" type="xslstylesheet.p" ignore-errors="false">
    <input name="xsl" label="book.xsl"/>
    <outparam name="stylesheet" label="xslstyle"/>
  </process>

  <process id="p3" type="xslprocess.p" ignore-errors="false">
    <param name="stylesheet" label="xslstyle"/>
  </process>

</pipeline>
```

Table 9–5 (Cont.)  orapipe Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-attr name value</td>
<td>Sets the value of $name to the specified value. For example, if the attribute name is source and the value is book.xml, then you can pass this value to an element in the pipeline document as follows: &lt;input ... label=&quot;$source&quot;&gt;.</td>
</tr>
</tbody>
</table>

Using the XML Pipeline Processor for Java  9-9
Processing XML in a Pipeline

Processes Specified in the Pipeline Document

In Example 9–1, three processes are called and associated with Java classes in the oracle.xml.pipeline.processes package. The pipeline document uses the <processdef/> element to make the following associations:

- domparser.p is associated with the DOMParserProcess class
- xslstylesheet.p is associated with the XSLStylesheetProcess class
- xslprocess.p is associated with the XSLProcess class

Processing Architecture Specified in the Pipeline Document

The PipelineSample program accepts the pipedoc.xml document shown in Example 9–1 as input along with XML documents book.xml and book.xsl. The basic design of the pipeline is as follows:

1. Parse the incoming book.xml document and generate a DOM tree. This task is performed by DOMParserProcess.
2. Parse book.xsl as a stream and generate an XSLStylesheet object. This task is performed by XSLStylesheetProcess.
3. Receive the DOM of book.xml as input, apply the stylesheet object, and write the result to myresult.html. This task is performed by XSLProcess.

Note the following aspects of the processing architecture used in the pipeline document:

- The target information set, http://example.org/myresult.html, is inferred from the default value of the target parameter and the xml:base setting.
- The process p2 has an input of book.xsl and an output parameter with the label xslstyle, so it has to run to produce the input for p3.
- The p3 process depends on input parameter xslstyle and document xmldoc.
- The p3 process has an output parameter with the label http://example.org/myresult.html, so it has to run to produce the target.
- The process p1 depends on input document book.xml and outputs xmldoc, so it has to run to produce the input for p3.
In Example 9–1, more than one order of processing can satisfy all of the dependencies. Given the rules, the XML Pipeline processor must process p3 last but can process p1 and p2 in either order or process them in parallel.

Writing a Pipeline Processor Application

The PipelineSample.java source file illustrates a basic pipeline application. You can use the application with any of the pipeline documents in Table 9–4 to parse and transform an input XML document.

The basic steps of the program are as follows:

1. Perform the initial setup. The program declares references of type FileReader (for the input XML file), PipelineDoc (for the input pipeline document), and PipelineProcessor (for the processor). The first argument is the pipeline document, which is required. If a second argument is received, then it is stored in the logname String. The following code fragment illustrates this technique:

   ```java
   public static void main(String[] args)
   {
       FileReader f;
       PipelineDoc pipe;
       PipelineProcessor proc;
       
       if (args.length < 1)
           
       if (args.length > 1)
           logname = args[1];
           ...
   }
   ...
   ```

2. Create a FileReader object by passing the first command-line argument to the constructor as the filename. For example:

   ```java
   f = new FileReader(args[0]);
   ```

3. Create a PipelineDoc object by passing the reference to the FileReader object. The following example casts the FileReader to a Reader and specifies no validation:

   ```java
   pipe = new PipelineDoc((Reader)f, false);
   ```

4. Instantiate an XML Pipeline processor. The following statement instantiates the pipeline processor:

   ```java
   proc = new PipelineProcessor();
   ```

5. Set the processor to the pipeline document. For example:

   ```java
   proc.setPipelineDoc(pipe);
   ```

6. Set the execution mode for the processor and perform any other configuration. The following code fragment uses a condition to determine the execution mode. If three or more arguments are passed to the program, then it sets the mode to sequential or parallel depending on which argument is passed. For example:

   ```java
   String execMode = null;
   if (args.length > 2)
   {
   ```
Processing XML in a Pipeline

```
execMode = args[2];
if(execMode.startsWith("seq"))
    proc.setExecutionMode(PipelineConstants.PIPELINE_SEQUENTIAL);
else if (execMode.startsWith("para"))
    proc.setExecutionMode(PipelineConstants.PIPELINE_PARALLEL);
}
```

7. Instantiate an error handler. The error handler must implement the 
   PipelineErrorHandler interface. The program uses the PipelineSampleErrHdler 
   shown in PipelineSampleErrHdlr.java. The following code fragment illustrates 
   this technique:
   ```java
   errHandler = new PipelineSampleErrHdlr(logname);
   ```

8. Set the error handler for the processor by invoking setErrorHandler(). The 
   following statement illustrates this technique:
   ```java
   proc.setErrorHandler(errHandler);
   ```

9. Execute the pipeline. The following statement illustrates this technique:
   ```java
   proc.executePipeline();
   ```

See Also: Oracle Database XML Java API Reference to learn about 
the oracle.xml.pipeline subpackages

Writing a Pipeline Error Handler

An application calling the XML Pipeline processor must implement the 
PipelineErrorHandler interface to handle errors received from the processor. Set the 
error handler in the processor by calling setErrorHandler(). When writing the error 
handler, you can choose to throw an exception for different types of errors.

The oracle.xml.pipeline.controller.PipelineErrorHandler interface declares the 
methods shown in Table 9–6, all of which return void.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error(java.lang.String msg, PipelineException e)</td>
<td>Handles PipelineException errors.</td>
</tr>
<tr>
<td>fatalError(java.lang.String msg, PipelineException e)</td>
<td>Handles fatal PipelineException errors.</td>
</tr>
<tr>
<td>warning(java.lang.String msg, PipelineException e)</td>
<td>Handles PipelineException warnings.</td>
</tr>
<tr>
<td>info(java.lang.String msg)</td>
<td>Prints optional, additional information about errors.</td>
</tr>
</tbody>
</table>

The first three methods in Table 9–6 receive a reference to an 
oracle.xml.pipeline.controller.PipelineException object. The following methods 
of the PipelineException class are especially useful:

- getExceptionType(), which obtains the type of exception thrown
- getProcessId(), which obtains the process ID where the exception occurred
- getMessage(), which returns the message string of this Throwable error

The PipelineSampleErrHdlr.java source file implements a basic error handler for 
use with the PipelineSample program. The basic steps are as follows:
1. Implement a constructor. The constructor accepts the name of a log file and wraps it in a PrintWriter object as follows:

```java
PipelineSampleErrHdlr(String logFile) throws IOException {
    log = new PrintWriter(new FileWriter(logFile));
}
```

2. Implement the `error()` method. This implementation prints the process ID, exception type, and error message. It also increments a variable holding the error count. For example:

```java
public void error (String msg, PipelineException e) throws Exception {
    log.println("\nError in: " + e.getProcessId());
    log.println("Type: ' + e.getExceptionType());
    log.println("Message: ' + e.getMessage());
    log.println("Error message: ' + msg);
    log.flush();
    errCount++;
}
```

3. Implement the `fatalError()` method. This implementation follows the pattern of `error()`. For example:

```java
public void fatalError (String msg, PipelineException e) throws Exception {
    log.println("\nFatalError in: " + e.getProcessId());
    log.println("Type: ' + e.getExceptionType());
    log.println("Message: ' + e.getMessage());
    log.println("Error message: ' + msg);
    log.flush();
    errCount++;
}
```

4. Implement the `warning()` method. This implementation follows the basic pattern of `error()` except it increments the `warnCount` variable rather than the `errCount` variable. For example:

```java
public void warning (String msg, PipelineException e) throws Exception {
    log.println("\nWarning in: " + e.getProcessId());
    log.println("Message: ' + e.getMessage());
    log.println("Error message: ' + msg);
    log.flush();
    warnCount++;
}
```

5. Implement the `info()` method. Unlike the preceding methods, this method does not receive a PipelineException reference as input. The following implementation prints the String received by the method and increments the value of the `warnCount` variable:

```java
public void info (String msg) {
    log.println("\nInfo : ' + msg);
    log.flush();
    warnCount++;
}
```
6. Implement a method to close the PrintWriter. The following code implements the method `closeLog()`, which prints the number of errors and warnings and calls `PrintWriter.close()`:

```java
public void closeLog()
{
    log.println("\nTotal Errors: " + errCount + "\nTotal Warnings: " +
               warnCount);
    log.flush();
    log.close();
}
```

**See Also:** *Oracle Database XML Java API Reference* to learn about the `PipelineErrorHandler` interface and the `PipelineException` class.
This chapter contains these topics:

- Introduction to XDK JavaBeans
- Using the XDK JavaBeans: Overview
- Processing XML with the XDK JavaBeans

Introduction to XDK JavaBeans

The Oracle XML JavaBeans are a set of XML components that you can use in Java applications and applets.

This section contains the following topics:

- Prerequisites
- Standards and Specifications
- XDK JavaBeans Features

Prerequisites

This chapter assumes that you are familiar with the following technologies:

- **JavaBeans**. JavaBeans components, or Beans, are reusable software components that can be manipulated visually in a builder tool.

- **Java Database Connectivity (JDBC)**. Database connectivity is included with the XDK JavaBeans. The beans can connect directly to a JDBC-enabled database to retrieve and store XML and XSL files.

- **Document Object Model (DOM)**. DOM is an in-memory tree representation of the structure of an XML document.

- **Simple API for XML (SAX)**. SAX is a standard for event-based XML parsing.

- **XML Schema language**. Refer to Chapter 7, "Using the Schema Processor for Java" for an overview and links to suggested reading.

Standards and Specifications

The XDK JavaBeans require version 1.2 or higher of the XDK and can be used with any version of JDK 1.2 or above. All of the XDK beans conform to the Sun JavaBeans specification and include the required `BeanInfo` class that extends `java.beans.SimpleBeanInfo`. 
The JavaBeans 1.01 specification, which describes JavaBeans as present in JDK 1.1, is available at the following URL:


The additions for the Java 2 platform to the JavaBeans core specification provide developers with standard means to create more sophisticated JavaBeans components. The JavaBeans specifications for Java 2 are available at the following URL:


See Also: Chapter 31, ”XDK Standards” for a summary of the standards supported by the XDK

XDK JavaBeans Features

The Oracle XDK JavaBeans facilitate the addition of GUIs to XML applications. Bean encapsulation includes documentation and descriptors that you can access directly from Java IDEs such as Oracle JDeveloper.

The XDK includes the following beans:

- DOMBuilder
- XSLTransformer
- DBAccess
- XMLDiff
- XMLCompress
- XMLDBAccess
- XSDValidator

DOMBuilder

The oracle.xml.async.DOMBuilder bean constructs a DOM tree from an XML document. The DOMBuilder JavaBean encapsulates the XML parser for Java DOMParser class with a bean interface and enhances by supporting asynchronous parsing. By registering a listener, Java programs can initiate parsing of large or successive documents and immediately return control to the caller.

One of the main benefits of this bean is increased efficiency when parsing multiple files, especially if the files are large. DOMBuilder can also provide asynchronous parsing in a background thread in interactive visual applications. Without asynchronous parsing, the GUI is useless until the document to be parsed. With DOMBuilder, the application calls the parse method and then resumes control. The application can display a progress bar, allow the user to cancel the parse, and so forth.

See Also: "Using the DOMBuilder JavaBean: Basic Process" on page 10-5

XSLTransformer

The oracle.xml.async.XSLTransformer bean supports asynchronous transformation. It accepts an XML document, applies an XSLT stylesheet, and creates an output file. The XSLTransformer JavaBean enables you to transform an XML document to almost any text-based format, including XML, HTML, and DDL. This bean can also be used as the basis of a server-side application or servlet to render an XML document, such as an XML representation of a query result, into HTML for display in a browser.
The main benefit of the XSLTransformer bean is that it can transform multiple files in parallel. Like DOMBuilder, you can also use it in visual applications to avoid long periods of time when the GUI is nonresponsive. By implementing the XSLTransformerListener interface, the calling application receives notification when the transformation completes.

**See Also:** "Using the XSLTransformer JavaBean: Basic Process" on page 10-7

**DBAccess**

The oracle.xml.dbaccess.DBAccess bean maintains CLOB tables that contain multiple XML and text documents. You can use it when you need to store and retrieve XML documents in the database, but do not need to process them within the database. Java applications that use the DBAccess bean connect to the database through JDBC. Note that XML documents stored in CLOB tables that are not of type XMLType do not have their entities expanded.

The DBAccess bean enables you to do perform the following tasks:

- Create and delete tables of type CLOB.
- Query the contents of CLOB tables.
- Perform INSERT, UPDATE, and DELETE operations on XML documents stored in CLOB tables.

**XMLDBAccess**

The oracle.xml.xmldbaccess.XMLDBAccess bean extends the DBAccess bean to support XML documents stored in XMLType tables. The class provides methods to list, delete, or retrieve XMLType instances and their tables. For example, the getXMLXPathTextData() method retrieves the value of an XPath expression from an XML document.

DBAccess JavaBean maintains XMLType tables that can hold multiple XML and text documents. Each XML or text document is stored as a row in the table. The table is created with the following SQL statement:

```
CREATE TABLE (FILENAME   CHAR( ) UNIQUE,
              FILEDATA   SYS.XMLType);
```

The FILENAME field holds a unique string used as a key to retrieve, update, or delete the row. Document text is stored in the FILEDATA field.

The XMLDBAccess bean performs the following tasks:

- Creates and deletes XMLType tables
- Lists the contents of an XMLType column
- Performs INSERT, UPDATE, and DELETE operations on XML documents stored in XMLType tables

**See Also:** "Using the XMLDBAccess JavaBean: Basic Process" on page 10-8

**XMLDiff**

When comparing XML documents, it is usually unhelpful to compare them character by character. Most XML comparisons are concerned with differences in structure and
significant textual content, not differences in whitespace. The Oracle XML Differ's XMLDiff bean performs the following useful tasks:

- Constructs and compares the DOM trees for two input XML documents and indicates whether the documents are different.
- Provides a graphical display of the differences between two XML files. Specifically, you can see node insert, delete, modify, or move.
- Generates an XSLT stylesheet that can convert one of the input XML documents into the other document.

The XMLDiff bean is especially useful in pipeline applications. For example, an application could update an XML document, compare it with a previous version of the document, and store the XSLT stylesheet that shows the differences between them.

See Also:

- Chapter 9, "Using the XML Pipeline Processor for Java"
- "Using the XMLDiff JavaBean: Basic Process" on page 10-10

**XMLCompress**

As explained in "Compressing XML" on page 4-42, the Oracle XML parser includes a compressor that can serialize XML document objects as binary streams. Although a useful tool, compression with XML parser has the following disadvantages:

- When XML data is deserialized, it must be reparsed.
- The encapsulation of XML data in tags greatly increases its size.

The Oracle XML XMLCompress bean is an encapsulation of the XML compression functionality. It provides the following advantages when serializing and deserializing XML:

- It encapsulates the method that serializes the DOM, which results in a stream.
- XML processors can regenerate the DOM from the compressed stream without reparsing the XML.

The bean supports compression and decompression of input XML parsed by DOMParser or SAXParser. DOM compression supports inputs of type XMLType, CLOB, and BLOB.

To use different parsing options, parse the document before input and then pass the XMLDocument object to the compressor bean. The compression factor is a rough value based on the file size of the input XML file and the compressed file. The limitation of the compression factor method is that it can only be used when the compression is performed with java.io.File objects as parameters.

**XSDValidator**

The Oracle XML Schemavalidator XSDValidator bean encapsulates the XSDValidator class and adds capabilities for validating a DOM tree. One of the most useful features of this bean concerns validation errors. If the application throws a validation error, the getStackList() method returns a list of DOM tree paths that lead to the invalid node. Nodes with errors are returned in a vector of stack trees in which the top element of the stack represents the root node. You can obtain child nodes by pulling them from the stack. To use getStackList() you must use instantiate the java.util.Vector and java.util.Stack classes.
Using the XDK JavaBeans: Overview

This section contains the following topics:

- Using the XDK JavaBeans: Basic Process
- Running the JavaBean Demo Programs

Using the XDK JavaBeans: Basic Process

This section describes the program flow of Java applications that use the more useful beans: DOMBuilder, XSLTransformer, XMLDBAccess, and XMLDiff. The section contains the following topics:

- Using the DOMBuilder JavaBean: Basic Process
- Using the XSLTransformer JavaBean: Basic Process
- Using the XMLDBAccess JavaBean: Basic Process
- Using the XMLDiff JavaBean: Basic Process

Using the DOMBuilder JavaBean: Basic Process

The DOMBuilder class implements an XML 1.0 parser according to the W3C recommendation. It parses an XML document and builds a DOM tree. The parsing is done in a separate thread. The DOMBuilderListener interface must be used for notification when the tree is built.

When developing applications that use this bean, you should import the following subpackages:

- oracle.xml.async, which provides asynchronous Java beans for DOM building
- oracle.xml.parser.v2, which provides APIs for SAX, DOM, and XSLT

The oracle.xml.parser.v2 subpackage is described in detail in Chapter 4, "XML Parsing for Java". The most important DOM-related classes and interfaces in the javax.xml.async package are described in Table 10–1.

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMBuilder class</td>
<td>Encapsulates an XML parser to parse an XML document and build a DOM tree. The parsing is done in a separate thread. The DOMBuilderListener interface must be used for notification when the tree is built.</td>
</tr>
<tr>
<td>DOMBuilderEvent class</td>
<td>Instantiates the event object that DOMBuilder uses to notify all registered listeners about parse events.</td>
</tr>
<tr>
<td>DOMBuilderListener interface</td>
<td>Must be implemented so that the program can receive notifications about events during the asynchronous parsing. The class implementing this interface must be added to the DOMBuilder with the addDOMBuilderListener() method.</td>
</tr>
<tr>
<td>DOMBuilderErrorEvent class</td>
<td>Defines the error event that is sent when parse exception occurs.</td>
</tr>
<tr>
<td>DOMBuilderErrorListener interface</td>
<td>Must be implemented so that the program can receive notifications when errors are found during parsing. The class implementing this interface must be added to the DOMBuilder with the addDOMBuilderErrorListener() method.</td>
</tr>
</tbody>
</table>

Figure 10–1 depicts the basic process of an application that uses the DOMBuilder JavaBean.
Figure 10–1 shows the following stages of XML processing:

1. Parse the input XML document. The program can receive the XML document as a file, string buffer, or URL.
2. Add the DOMBuilder listener. The program invokes the method `DOMBuilder.addDOMBuilderListener(DOMBuilderListener).`
3. Parse the XML document. The program invokes the `DOMBuilder.parse()` method.
4. Optionally, process the parsed result further.
5. Call the listener when the program receives an asynchronous call. The listener, which must implement the `DOMBuilderListener` interface, is called by invoking the `DOMBuilderOver()` method.

The available `DOMBuilderListener` methods are:
- `domBuilderError(DOMBuilderEvent)`, which is called when parse errors occur.
- `domBuilderOver(DOMBuilderEvent)`, which is called when parsing completes.
- `domBuilderStarted(DOMBuilderEvent)`, which is called when parsing begins.

6. Fetch the DOM. Invoke the `DOMBuilder.getDocument()` method to fetch the resulting DOM document and output it.
Using the XSLTransformer JavaBean: Basic Process

The XSLTransformer bean encapsulates the Java XML parser XSLT processing engine with a bean interface and extends its functionality to permit asynchronous transformation. By registering a listener, your Java application can transform large and successive documents by having the control returned immediately to the caller.

When developing applications that use this bean, you should import the following subpackages:

- `oracle.xml.async`, which provides asynchronous Java beans for XSL transformations
- `oracle.xml.parser.v2`, which provides APIs for XML parsing SAX, DOM, and XSLT

The `oracle.xml.parser.v2` subpackage is described in detail in Chapter 4, "XML Parsing for Java". The most important XSL-related classes and interfaces in the `javax.xml.async` package are described in Table 10–2.

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLTransformer class</td>
<td>Applies XSL transformation in a background thread.</td>
</tr>
<tr>
<td>XSLTransformerEvent class</td>
<td>Represents the event object used by XSLTransformer to notify XSL transformation events to all of its registered listeners.</td>
</tr>
<tr>
<td>XSLTransformerListener interface</td>
<td>Must be implemented so that the program can receive notifications about events during asynchronous transformation. The class implementing this interface must be added to the XSLTransformer with the addXSLTransformerListener() method.</td>
</tr>
<tr>
<td>XSLTransformerErrorEvent class</td>
<td>Instantiates the error event object that XSLTransformer uses to notify all registered listeners about transformation error events.</td>
</tr>
<tr>
<td>XSLTransformerErrorListener interface</td>
<td>Must be implemented so that the program can receive notifications about error events during the asynchronous transformation. The class implementing this interface must be added to the XSLTransformer using addXSLTransformerListener() method.</td>
</tr>
</tbody>
</table>

Figure 10–2 illustrates XSLTransformer bean usage.
Figure 10–2  XSLTransformer JavaBean Usage

Figure 10–2 goes through the following stages:

1. Input an XSLT stylesheet and XML instance document.
2. Add an XSLT listener. The program invokes the XSLTransformer.addXSLTransformerListener() method.
3. Apply the stylesheets. The XSLTransformer.processXSL() method initiates the XSL transformation in the background.
4. Optionally, perform further processing with the XSLTransformer bean.
5. Call the XSLT listener when the program receives an asynchronous call. The listener, which must implement the XSLTransformerListener interface, is called by invoking the xslTransformerOver() method.
6. Fetch the result of the transformation. Invoke the XSLTransformer.getResult() method to return the XML document fragment for the resulting document.
7. Output the XML document fragment.

Using the XMLDBAccess JavaBean: Basic Process

When developing applications that use the XMLDBAccess bean, you should use the following subpackages:

- oracle.xml.xmldbaccess, which includes the XMLDBAccess bean
- oracle.xml.parser.v2, which provides APIs for XML parsing SAX, DOM, and XSLT

The oracle.xml.parser.v2 subpackage is described in detail in Chapter 4, "XML Parsing for Java". Some of the more important methods in the XMLDBAccess class are described in Table 10–3.
Using the XDK JavaBeans: Overview

Figure 10–3 illustrates typical XMLDBAccess bean usage. It shows how the DBAccess bean maintains and manipulates XML documents stored in XMLTypes.

For example, an XMLDBAccess program could process XML documents in the following stages:

1. Create an XMLType table. Invoke `createXMLTypeTable()` by passing it database connection information and a table name.
2. List the XMLType tables. Invoke `getXMLTypeTableNames()` by passing it database connection information and an empty string.
3. Load XML data into the table. Invoke `replaceXMLTypeData()` by passing it database connection information, the table name, the XML file name, and a string containing the XML.
4. Retrieve the XML data into a `String`. Invoke `getXMLTypeData()` by passing it the connection information, the table name, and the XML file name.
5. Retrieve XML data based on an XPath expression. Invoke `getXMLXPathTextData()` by passing it the connection information, the table name, the XML file name, and the XPath expression.
6. Close the connection.
Using the XMLDiff JavaBean: Basic Process

When developing applications that use the XMLDiff bean, you typically use the following subpackages:

- oracle.xml.xmldiff, which includes the XMLDiff bean
- oracle.xml.parser.v2, which provides APIs for XML parsing SAX, DOM, and XSLT
- oracle.xml.async, which provides asynchronous Java beans for DOM building

The oracle.xml.parser.v2 subpackage is described in detail in Chapter 4, "XML Parsing for Java". Some important methods in the XMLDiff class are described in Table 10–4.

Table 10–4 XMLDiff Methods

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff()</td>
<td>Determines the differences between two input XML files or two XMLDocument objects.</td>
</tr>
<tr>
<td>generateXSL()</td>
<td>Generates an XSL file that represents the differences between the input two XML files. The first XML file can be transformed into the second XML file with the generated stylesheet. If the XML files are the same, then the XSL generated can transform the first XML file into the second XML file, where the first and second files are equivalent. Related methods are generateXSLDoc() and generateXSLFile().</td>
</tr>
<tr>
<td>setFiles()</td>
<td>Sets the XML files that need to be compared.</td>
</tr>
<tr>
<td>getDocument1()</td>
<td>Gets the document root as an XMLDocument object of the first XML tree. getDocument2() is the equivalent method for the second tree.</td>
</tr>
<tr>
<td>getDiffPanel1()</td>
<td>Gets the text panel as JTextPane object that visually shows the diffs in the first XML file. getDiffPanel2() is the equivalent method for the second file.</td>
</tr>
</tbody>
</table>

Figure 10–4 illustrates typical XMLDiff bean usage. It shows how XMLDiff bean compares and displays the differences between input XML documents.
For example, an XMLDiff program could process XML documents in the following stages:

1. Create an XMLDiff object.
2. Set the files to be compared. Create File objects for the input files and pass references to the objects to XMLDiff.setFiles().
3. Compare the documents. The diff() method returns false if the XML files are the same and true if they are different.
4. Respond depending on the whether the input XML documents are the same or different. For example, if they are the same, invoke JOptionPane.showMessageDialog() to print a message.
5. Generate an XSLT stylesheet that shows the differences between the input XML documents. For example, generateXSLDoc() generates an XSL stylesheet as an XMLDocument.
6. Display the DOM trees created by XMLDiff.

Running the JavaBean Demo Programs

Demo programs for the XDK JavaBeans are included in the $ORACLE_HOME/xdk/demo/java/transviewer directory. The demos illustrate the use of the XDK beans described in "XDK JavaBeans Features" on page 10-2 as well as some visual
beans that are now deprecated. The following list shows all of the beans used in the demos:

- XSLTransformer
- DOMBuilder
- DBAccess
- XMLDBAccess
- XMLDiff
- XMLCompress
- XSDValidator
- oracle.xml.srcviewer.XMLSourceView (deprecated)
- oracle.xml.treeviewer.XMLTreeView (deprecated)
- oracle.xml.transformer.XMLTransformPanel (deprecated)
- oracle.xml.dbviewer.DBViewer (deprecated)

Although the visual beans are deprecated, they remain useful as educational tools. Consequently, the deprecated beans are included in $ORACLE_HOME/lib/xmldemo.jar. The nondeprecated beans are included in $ORACLE_HOME/lib/xml.jar.

Table 10–5 lists the sample programs provided in the demo directory. The first column of the table indicates which sample program use deprecated beans.

<table>
<thead>
<tr>
<th>Sample</th>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample1</td>
<td>XMLTransformPanelSample.java</td>
<td>A visual application that uses the XMLTransformPanel, DOMBuilder, and XSLTransformer beans. This bean applies XSL transformations to XML documents and shows the result. See Also: “Running sample1” on page 10-15</td>
</tr>
<tr>
<td>sample2</td>
<td>ViewSample.java</td>
<td>A sample visual application that uses the XMLSourceView and XMLTreeView beans. It visualizes XML document files. See Also: “Running sample2” on page 10-15</td>
</tr>
<tr>
<td>sample3</td>
<td>AsyncTransformSample.java</td>
<td>A nonvisual application that uses the XSLTransformer and DOMBuilder beans. It applies the XSLT stylesheet specified in doc.xsl on all .xml files in the current directory. It writes the results to files with the extension .log. See Also: “Running sample3” on page 10-15</td>
</tr>
<tr>
<td>sample4</td>
<td>DBViewSample.java</td>
<td>A visual application that uses the DBViewer bean to implement a simple application that handles insurance claims. See Also: “Running sample4” on page 10-15</td>
</tr>
<tr>
<td></td>
<td>DBViewClaims.java</td>
<td>This JFrame subclass is instantiated in the DBViewFrame class, which is in turn instantiated in the DBViewSample program.</td>
</tr>
<tr>
<td></td>
<td>DBViewFrame.java</td>
<td>This JFrame subclass is instantiated in the DBViewSample program.</td>
</tr>
<tr>
<td>sample5</td>
<td>XMLDBAccessSample.java</td>
<td>A nonvisual application for the XMLDBAccess bean. This program demonstrates how to use the XMLDBAccess bean APIs to store and retrieve XML documents in XMLType tables. To use XMLType, an Oracle database is necessary along with xdb.jar. The program accepts values for HOSTNAME, PORT, SID, USERID, and PASSWORD. The program creates tables in the database and loads data from file booklist.xml. The program writes output to xmldbaccess.log. See Also: “Running sample5” on page 10-16</td>
</tr>
</tbody>
</table>
Table 10–6 describes additional files that are used by the demo programs.

<table>
<thead>
<tr>
<th>Sample</th>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample6</td>
<td>XMLDiffSample.java</td>
<td>A visual application that uses the XMLDiff bean to find differences between two XML files and generate an XSLT stylesheet. You can use this stylesheet to transform the first input XML into the second input XML file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>See Also:</strong> &quot;Running sample6&quot; on page 10-17</td>
</tr>
<tr>
<td></td>
<td>XMLDiffFrame.java</td>
<td>A class that implements the ActionListener interface. This class is used by the XMLDiffSample program.</td>
</tr>
<tr>
<td></td>
<td>XMLDiffSrcView.java</td>
<td>A JPanel subclass used by the XMLDiffSample program.</td>
</tr>
<tr>
<td>sample7</td>
<td>compviewer.java</td>
<td>A visual application that uses the XMLCompress bean to compress XML. The XML input can be an XML file or XML data obtained through a SQL query. The application enables you to decompress the compressed stream and view the resulting DOM tree.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>See Also:</strong> &quot;Running sample7&quot; on page 10-17</td>
</tr>
<tr>
<td></td>
<td>compstreamdata.java</td>
<td>A simple class that pipes information from the GUI to the bean. This class is used in dbpanel.java, filepanel.java, and xmlcompressutil.java.</td>
</tr>
<tr>
<td></td>
<td>dbpanel.java</td>
<td>A JPanel subclass used in xmlcompressutil.java.</td>
</tr>
<tr>
<td></td>
<td>filepanel.java</td>
<td>A JPanel subclass used in xmlcompressutil.java.</td>
</tr>
<tr>
<td></td>
<td>xmlcompressutil.java</td>
<td>A JPanel subclass used in compviewer.java.</td>
</tr>
<tr>
<td>sample8</td>
<td>XMLSchemaTreeViewer.java</td>
<td>A visual application that uses the Treeviewer, sourceviewer, and XSDValidator beans. The application accepts an XML instance document and an XML schema document as inputs. The application parses both the documents and validates the instance document against the schema. If the document is invalid, then the nodes where the errors occurred are highlighted and an error message is shown in a tool tip.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>See Also:</strong> &quot;Running sample8&quot; on page 10-17</td>
</tr>
<tr>
<td></td>
<td>TreeViewerValidate.java</td>
<td>A JPanel subclass that displays a parsed XML instance document as a tree. This class is used by the XMLSchemaTreeViewer.java program.</td>
</tr>
<tr>
<td>sample9</td>
<td>XMLSrcViewer.java</td>
<td>A visual application that uses the sourceviewer and XSDValidator beans. The demo takes an XML file as input. You can select the validation mode: DTD, XML schema, or no validation. The program validates the XML data file against the DTD or schema and displays it with syntax highlighting. It also logs validation errors. For schema validation it also highlights the error nodes appropriately. External and internal DTDs can be viewed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>See Also:</strong> &quot;Running sample9&quot; on page 10-18</td>
</tr>
<tr>
<td></td>
<td>XMLSrcViewPanel.java</td>
<td>A class that shows how to use the XMLSourceView and DTDSourceView objects. This class is used by the XMLSrcViewer.java program. Each XMLSourceView object is set as a Component of a JPanel by invoking goButton_actionPerformed(). The XML file to be viewed is parsed and the resulting XML document is set in the XMLSourceView object by invoking makeSrcPane(). The highlighting and DTD display properties are specified at this time. For performing schema validation, build the schema object by invoking makeSchemaValPane(). You can check for errors and display the source code accordingly with different highlights. You can retrieve a list of schema validation errors from the XMLSourceView by invoking dumpErrors().</td>
</tr>
<tr>
<td>sample10</td>
<td>XSDValidatorSample.java</td>
<td>An application that shows how to use the XSDValidator bean. It accepts an XML file and an XML schema file as input. The program displays errors occurring during validation, including line numbers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>See Also:</strong> &quot;Running sample10&quot; on page 10-18</td>
</tr>
</tbody>
</table>
Documentation for how to compile and run the sample programs is located in the README in the same directory. The basic steps are as follows:

1. Change into the `$/ORACLE_HOME/xdk/demo/java/transviewer` directory (UNIX) or `%ORACLE_HOME%/xdk/demo/java\transviewer` directory (Windows).

2. Make sure that your environment variables are set as described in "Setting Up the Java XDK Environment" on page 3-5. The beans require JDK 1.2 or higher. The DBViewer and DBTransformPanel beans require JDK 1.2.2 when rendering HTML. Prior versions of the JDK may not render HTML in the result buffer properly.

3. Edit the Makefile (UNIX) or Make.bat (Windows) for your environment. In particular, do the following:
   - Change the JDEPATH in the Makefile to point to your JDK path.
   - Change PATHSEP to the appropriate path separator for your operating system.
   - Change the HOSTNAME, PORT, SID, USERID, and PASSWORD parameters so that you can connect to the database through the JDBC thin driver. These parameters are used in `sample4` and `sample5`.

4. Run `make` (UNIX) or `Make.bat` (Windows) at the system prompt to generate the class files.

5. Run `gmake` as follows to run the demos:

   ```
gmake sample1
gmake sample2
gmake sample3
gmake sample4
gmake sample5
gmake sample6
```
The following sections explain how to run the demos.

Running sample1

Sample1 is the program that uses the XMLTransViewer bean. You can run the program manually as follows:

```
java XMLTransformPanelSample
```

You can use the program to import and export XML files from Oracle database, store XSL transformation files in the database, and apply stylesheets to XML interactively. To use the database connectivity feature in this program, you need to know the network name of the computer where the database runs, the port (usually 1521), and the name of the Oracle instance (usually orcl). You also need an account with CREATE TABLE privileges. If you have installed the sample schemas, then you can use the account hr. You can use the XMLTransViewer program to apply stylesheet transformation to XML files and display the result.

The program displays a panel with tabs on the top and the bottom. The first two top tabs are used to switch between the XML buffer and the XSLT buffer. The third tab performs XSL transformation on the XML buffer and displays the result. The first two tabs on the bottom can be used to load and save data from Oracle database and from the file system. The remaining bottom tabs switch the display of the current content to tree view, XML source, edit mode and, in case of the result view after the transformation, HTML.

Running sample2

Sample2 is a GUI-based demo for the XMLSourceView and XMLTreeView beans, which are deprecated. The ViewSample program displays the booklist.xml file in separate source and tree views. You can run the program manually as follows:

```
java ViewSample
```

Running sample3

Sample3 is a nonvisual demo for the asynchronous DOMBuilder and XSLTransformer beans. The AsyncTransformSample program applies the doc.xsl XSLT stylesheet to all *.xml files in the current directory. The program writes output to files with the extension .log. You can run the program as follows:

```
java AsyncTransformSample
```

Running sample4

Sample4 is a visual demo for the DBViewer bean, which is deprecated. It runs in the following stages:

1. It starts SQL*Plus, connects to the database with the USERID and PASSWORD specified in the Makefile, and runs the claim.sql script. This script creates a number of tables, views, and types for use by the DBViewSample demo program.

2. It runs the DBViewSample program as follows:

```
java -classpath "$(MAKE_CLASSPATH)" DBViewSample
```
JDBC connection information is hard-coded in the DBViewClaims.java source file, which implements a class used by the demo. Specifically, the program assumes the values for USERID, PASSWORD, and so forth set in the Makefile. If your configuration is different, navigate to line 92 in DBViewClaims.java and modify setUsername(), setPassword(), and so forth with values that reflect your Oracle database configuration.

Running sample5

Sample5 is a nonvisual demo for the XMLDBAccess bean. It uses the XMLType objects to store XML documents inside the database. The following program connects to the database with the Java thin client, creates XMLType tables, and loads the data from booklist.xml. To run the program you must specify the following pieces of information as command-line arguments:

- Host name (for example, myhost)
- Port number (for example, 1521)
- SID of the database (for example, ORCL)
- Database account in which the tables will be created (for example, hr)
- Password for the database account (for example, hr)

For example, you can run the program as follows:

```java
java XMLDBAccessSample myhost 1521 ORCL hr hr
```

The following text shows sample output from dbaccess.log:

Demo for createXMLTypeTables():
Table +'testxmltype' successfully created.

Demo for listXMLTypeTables():
tablenamename=TESTXMLTYPE

Demo for replaceXMLTypeData() (similar to insert):
XML Data from +'booklist.xml' successfully replaced in table 'testxmltype'.

Demo for getXMLTypeData():
XMLType data fetched:
<?xml version="1.0"?>
<booklist>
  <book isbn="1234-123456-1234">
    <title>C Programming Language</title>
    <author>Kernighan and Ritchie</author>
    <publisher>EEE</publisher>
    <price>7.99</price>
  </book>
  ...
  <book isbn="1230-23498-2349879">
    <title>Emperor's New Mind</title>
    <author>Roger Penrose</author>
    <publisher>Oxford Publishing Company</publisher>
    <price>15.99</price>
  </book>
</booklist>

Demo for getXMLTypeXPathTextData():
Data fetched using XPath exp '/booklist/book[3]':
<br isbn="2137-598354-65978">
Running sample6
The sample6 program is a visual demo for the XMLDiff bean. The XMLDiffSample class invokes a GUI that enables you to choose the input data files from the File menu by selecting Compare XML Files. The Transform menu enables you to apply the generated XSLT generated to the first input XML. Select Save As in the File menu to save the output XML file, which will be the same as the second input file. By default, the program compares XMLDiffData1.txt to XMLDiffData2.txt and stores the XSLT output as XMLDiffSample.xsl.

You can run the program manually as follows:
java -mx50m XMLDiffSample XMLDiffData1.txt XMLDiffData2.txt

If the input XML files use a DTD that accesses a URL outside a firewall, then modify XMLDiffSample.java to include the proxy server settings before the setFiles() call. For example, modify the program as follows:

```java
/* Set proxy to access dtd through firewall */
Properties p = System.getProperties();
p.put("proxyHost", "www.proxyservername.com");
p.put("proxyPort", "80");
p.put("proxySet", "true");
/* You will also have to import java.util.*; */
```

Running sample7
The sample7 visual demo illustrates the use of the XMLCompress bean. The compviewer class invokes a GUI which lets the user compress and uncompress XML files and data obtained from the database. The loading options enable the user to retrieve the data either from a file system or a database. This application does not support loading and saving compressed data from the database. The compression factor indicates a rough estimate by which the XML data is reduced.

You can run the program manually as follows:
java compviewer

Running sample8
The sample8 demo illustrates the use of the XMLTreeViewer bean. The XMLSchemaTreeViewer program enables the user to view an XMLDocument in a tree format. The user can input a schema document and validate the instance document against the schema. If the document is invalid, then the invalid nodes are highlighted with the error message. Also, the program displays a log of all the line information in a separate panel, which enables the user to edit the instance document and revaluated.
Test the program with sample files purchaseorder.xml and purchaseorder.xsd. The instance document purchaseorder.xml does not comply with schema defined in purchaseorder.xsd.

You can run the program manually as follows:
java XMLSchemaTreeViewer
Running sample9

The sample9 demo illustrates the use of the SourceViewer bean. The XMLSrcViewer program enables you to view an XML document or a DTD with syntax highlighting turned on. You can validate the XML document against an input XML Schema or DTD. The DTD can be internal or external.

If the validation is successful, then you can view the instance document and XML schema or DTD in the Source View pane. If errors were encountered during schema validation, then an error log with line numbers is available in the Error pane. The Source View pane shows the XML document with error nodes highlighted.

You can use sample files purchaseorder.xml and purchaseorder.xsd for testing XML schema validation with errors. You can use note_in_dtd.xml with DTD validation mode to view an internal DTD with validation errors. You can run the program manually as follows:

```
java XMLSrcViewer
```

Running sample10

The sample10 demo illustrates the use of the XSDValidator bean. The XSDValidatorSample program's two arguments are an XML document and its associated XML schema. The program displays errors occurring during validation, including line numbers.

The following program uses purchaseorder.xsd to validate the contents of purchaseorder.xml:

```
java XSDValidatorSample purchaseorder.xml purchaseorder.xsd
```

The XML document fails (intentionally) to validate against the schema. The program displays the following errors:

```
Sample purchaseorder.xml purchaseorder.xsd
<Line 2, Column 41>: XML-24523: (Error) Invalid value 'abc' for attribute: 'orderDate'
 #document->purchaseOrder
<Line 7, Column 27>: XML-24525: (Error) Invalid text 'CA' in element: 'state'
 #document->purchaseOrder->shipTo->state->#text
<Line 8, Column 25>: XML-24525: (Error) Invalid text 'sd' in element: 'zip'
 #document->purchaseOrder->shipTo->zip->#text
<Line 14, Column 27>: XML-24525: (Error) Invalid text 'PA' in element: 'state'
 #document->purchaseOrder->billTo->state->#text
<Line 17, Column 22>: XML-24534: (Error) Element 'coment' not expected.
 #document->purchaseOrder->coment
<Line 29, Column 31>: XML-24534: (Error) Element 'shipDae' not expected.
 #document->purchaseOrder->items->item->shipDae
```

Processing XML with the XDK JavaBeans

This section contains the following topics:

- Processing XML Asynchronously with the DOMBuilder and XSLTransformer Beans
- Comparing XML Documents with the XMLDiff Bean
Processing XML Asynchronously with the DOMBuilder and XSLTransformer Beans

As explained in "DOMBuilder" on page 10-2 and "XSLTransformer" on page 10-2, you can use XDK beans to perform asynchronous XML processing.

The AsyncTransformSample.java program illustrates how to use both the DOMBuilder and XSLTransformer beans. The program implements the following methods:

- runDOMBuilders()
- runXSLTransformer()
- saveResult()
- makeXSLDocument()
- createURL()
- init()
- exitWithError()
- asyncTransform()

The basic architecture of the program is as follows:

1. The program declares and initializes the fields used by the class. Note that the input XSLT stylesheet is hard-coded in the program as doc.xsl. The class defines the following fields:

   ```java
   String    basedir = new String(".");
   OutputStream  errors = System.err;
   Vector    xmlfiles = new Vector();
   int       numXMLDocs = 1;
   String    xslFile = new String("doc.xsl");
   URL       xslURL;
   XMLDocument xslDoc
   ```

2. The main() method invokes the init() method to perform the initial setup. This method lists the files in the current directory, and if it finds files that end in the extension .xml, it adds them to a Vector object. The implementation for the init() method is as follows:

   ```java
   boolean init () throws Exception
   {
     File     directory = new File(basedir);
     String[] dirfiles = directory.list();
     for (int j = 0; j < dirfiles.length; j++)
     {
       String dirfile = dirfiles[j];
       if (!dirfile.endsWith(".xml"))
         continue;
       xmlfiles.addElement(dirfile);
     }
     if (xmlfiles.isEmpty()) {
       System.out.println("No files in directory were selected for processing");
       return false;
     }
     numXMLDocs = xmlfiles.size();
     return true;
   }
   ```
3. The main() method instantiates AsyncTransformSample as follows:
   AsyncTransformSample inst = new AsyncTransformSample();

4. The main() method invokes the asyncTransform() method. The asyncTransform() method performs the following main tasks:
   a. Invokes makeXSLDocument() to parse the input XSLT stylesheet.
   b. Calls runDOMBuilders() to initiate parsing of the instance documents, that is, the documents to be transformed, and then transforms them.

After initiating the XML processing, the program resumes control and waits while the processing occurs in the background. When the last request completes, the method exits.

The following code shows the implementation of the asyncTransform() method:

```java
void asyncTransform () throws Exception
{
    System.err.println (numXMLDocs +
            " XML documents will be transformed" +
            " using XSLT stylesheet specified in " + xslFile +
            " with " + numXMLDocs + " threads");

    makeXSLDocument ();
    runDOMBuilders ();

    // wait for the last request to complete
    while (rm.activeFound())
    {
        Thread.sleep(100);
    }
}
```

The following sections explain the makeXSLDocument() and runDOMBuilders() methods.

**Parsing the Input XSLT Stylesheet**

The makeXSLDocument() method illustrates a simple DOM parse of the input stylesheet. It does not use asynchronous parsing. The technique is the same described in "Performing Basic DOM Parsing" on page 4-15.

The method follows these steps:

1. Create a new DOMParser() object. The following code fragment from DOMSample.java illustrates this technique:
   ```java
   DOMParser parser = new DOMParser();
   ```

2. Configure the parser. The following code fragment specifies that whitespace should be preserved:
   ```java
   parser.setPreserveWhitespace(true);
   ```

3. Create a URL object from the input stylesheet. The following code fragment invokes the createURL() helper method to accomplish this task:
   ```java
   xslURL = createURL (xslFile);
   ```

4. Parse the input XSLT stylesheet. The following statement illustrates this technique:
   ```java
   parser.parse (xslURL);
   ```
5. Obtain a handle to the root of the in-memory DOM tree. You can use the
XMLDocument object to access every part of the parsed XML document. The
following statement illustrates this technique:

```java
xsldoc = parser.getDocument();
```

### Processing the XML Documents Asynchronously

The runDOMBuilders() method illustrates how you can use the DOMBuilder and
XSLTransformer beans to perform asynchronous processing. The parsing and
transforming of the XML occurs in the background.

The method follows these steps:

1. Create a resource manager to manage the input XML documents. The program
creates a for loop and obtains the XML documents. The following code fragment
illustrates this technique:

```java
rm = new ResourceManager (numXMLDocs);
for (int i = 0; i < numXMLDocs; i++)
{
    rm.getResource();
    ...
}
```

2. Instantiate the DOM builder bean for each input XML document. For example:

```java
DOMBuilder builder = new DOMBuilder(i);
```

3. Create a URL object from the XML file name. For example:

```java
DOMBuilder builder = new DOMBuilder(i);
URL xmlURL = createURL(basedir + '/' + (String)xmlfiles.elementAt(i));
if (xmlURL == null)
    exitWithError("File " + (String)xmlfiles.elementAt(i) + " not found");
```

4. Configure the DOM builder. The following code fragment specifies the
preservation of whitespace and sets the base URL for the document:

```java
builder.setPreserveWhitespace(true);
builder.setBaseURL (createURL(basedir + '/'));
```

5. Add the listener for the DOM builder. The program adds the listener by invoking
addDOMBuilderListener().

The class instantiated to create the listener must implement the
DOMBuilderListener interface. The program provides a do-nothing
implementation for domBuilderStarted() and domBuilderError(), but must
provide a substantive implementation for domBuilderOver(), which is the method
called when the parse of the XML document completes. The method invokes
runXSLTransformer(), which is the method that transforms the XML. Refer to
"Transforming the XML with the XSLTransformer Bean" on page 10-22 for an
explanation of this method.

The following code fragment illustrates how to add the listener:

```java
builder.addDOMBuilderListener
{ new DOMBuilderListener()
{
    public void domBuilderStarted(DOMBuilderEvent p0) {}
    public void domBuilderError(DOMBuilderEvent p0) {}
    public synchronized void domBuilderOver(DOMBuilderEvent p0)
```
{ DOMBuilder bld = (DOMBuilder)p0.getSource();
runXSLTransformer (bld.getDocument(), bld.getId());
}

6. Add the error listener for the DOM builder. The program adds the listener by invoking addDOMBuilderErrorListener(). The class instantiated to create the listener must implement the DOMBuilderErrorListener interface. The following code fragment show the implementation:

```java
builder.addDOMBuilderErrorListener
{
   new DOMBuilderErrorListener()
   {
      public void domBuilderErrorCalled(DOMBuilderErrorEvent p0)
      {
         int id = ((DOMBuilder)p0.getSource()).getId();
exitWithError("Error occurred while parsing " +
xmlfiles.elementAt(id) + ": " +
p0.getException().getMessage());
      }
   }
};
```

7. Parse the document. The following statement illustrates this technique:

```java
builder.parse (xmlURL);
System.err.println("Parsing file " + xmlfiles.elementAt(i));
```

Translating the XML with the XSLTransformer Bean When the DOM parse completes, the DOM listener receives notification. The domBuilderOver() method implements the behavior in response to this event. The program passes the DOM to the runXSLTransformer() method, which initiates the XSL transformation.

The method follows these steps:

1. Instantiate the XSLTransformer bean. This object performs the XSLT processing. The following statement illustrates this technique:

   ```java
   XSLTransformer processor = new XSLTransformer (id);
   ```

2. Create a new stylesheet object. For example:

   ```java
   XSLStylesheet  xsl       = new XSLStylesheet (xsldoc, xslURL);
   ```

3. Configure the XSLT processor. For example, the following statement sets the processor to show warnings and configures the error output stream:

   ```java
   processor.showWarnings (true);
   processor.setErrorStream (errors);
   ```

4. Add the listener for the XSLT processor. The program adds the listener by invoking addXSLTransformerListener(). The class instantiated to create the listener must implement the XSLTransformerListener interface. The program provides a do-nothing implementation for xslTransformerStarted() and xslTransformerError(), but
must provide a substantive implementation for `xslTransformerOver()`, which is the method called when the parse of the XML document completes. The method invokes `saveResult()`, which prints the transformation result to a file.

The following code fragment illustrates how to add the listener:

```java
processor.addXSLTransformerListener
{
    new XSLTransformerListener()
    {
        public void xslTransformerStarted (XSLTransformerEvent p0) {}
        public void xslTransformerError(XSLTransformerEvent p0) {}
        public void xslTransformerOver (XSLTransformerEvent p0)
        {
            XSLTransformer trans = (XSLTransformer)p0.getSource();
            saveResult (trans.getResult(), trans.getId());
        }
    }
};
```

5. Add the error listener for the XSLT processor. The program adds the listener by invoking `addXSLTransformerErrorListener()`.

The class instantiated to create the listener must implement the `XSLTransformerErrorListener` interface. The following code fragment shows the implementation:

```java
processor.addXSLTransformerErrorListener
{
    new XSLTransformerErrorListener()
    {
        public void xslTransformerErrorCalled(XSLTransformerErrorEvent p0)
        {
            int i = ((XSLTransformer)p0.getSource()).getId();
            exitWithError("Error occurred while processing " +
            xmlfiles.elementAt(i) + ": " +
            p0.getException().getMessage());
        }
    }
};
```

6. Transform the XML document with the XSLT stylesheet. The following statement illustrates this technique:

```java
processor.processXSL(xsl, xml);
```

### Comparing XML Documents with the XMLDiff Bean

As explained in “XMLDiff” on page 10-3, you can use XDK beans to compare the structure and significant content of XML documents.

The `XMLDiffSample.java` program illustrates how to use the `XMLDiff` bean. The program implements the following methods:

- `showDiffs()`
- `doXSLTransform()`
- `createURL()`

The basic architecture of the program is as follows:
1. The program declares and initializes the fields used by the class. Note that one field is of type XMLDiffFrame, which is the class implemented in the XMLDiffFrame.java demo. The class defines the following fields:

```java
protected XMLDocument doc1;   /* DOM tree for first file */
protected XMLDocument doc2;   /* DOM tree for second file */
protected static XMLDiffFrame diffFrame; /* GUI frame */
protected static XMLDiffSample dfxApp; /* XMLDiff sample application */
protected static XMLDiff xmlDiff; /* XML diff object */
protected static XMLDocument xslDoc; /* parsed xsl file */
protected static String outFile = new String("XMLDiffSample.xsl"); /* output xsl file name */
```

2. The main() method creates an XMLDiffSample object as follows:

```java
dfxApp = new XMLDiffSample();
```

3. The main() method adds and initializes a JFrame to display the output of the comparison. The following code illustrates this technique:

```java
diffFrame = new XMLDiffFrame(dfxApp);
diffFrame.addTransformMenu();
```

4. The main() method instantiates the XMLDiff bean. The following code illustrates this technique:

```java
xmlDiff = new XMLDiff();
```

5. The main() method invokes the showDiffs() method. This method performs the following tasks:

   a. Invokes XMLDiff.diff() to compare the input XML documents.

   b. Generates and displays an XSLT stylesheet that can transform one input document into the other document.

   The following code fragment shows the showDiffs() method call:

```java
if (args.length == 3)
    outFile = args[2];
if (args.length >= 2)
    dfxApp.showDiffs(new File(args[0]), new File(args[1]));
diffFrame.setVisible(true);
```

The following section explains the showDiffs() method.

### Comparing the XML Files and Generating a Stylesheet

The showDiffs() method illustrates the use of the XMLDiff bean.

The method follows these steps:

1. Set the files for the XMLDiff processor. The following statement illustrates this technique:

   ```java
   xmlDiff.setFiles(file1, file2);
   ```

2. Compare the files. The diff() method returns a boolean value that indicates whether the input documents have identical structure and content. If they are equivalent, then the method prints a message to the JFrame implemented by the XMLDiffFrame class. The following code fragment illustrates this technique:

   ```java
   if (!xmlDiff.diff())
   { 
   ```
null

JOptionPane.showMessageDialog
{
    diffFrame,
    "Files are equivalent in XML representation",
    "XMLDiffSample Message",
    JOptionPane.PLAIN_MESSAGE
};
}

3. Generate a DOM for the XSLT stylesheet that shows the differences between the two documents. The following code fragment illustrates this technique:

```java
xslDoc = xmlDiff.generateXSLDoc();
```

4. Display the documents in the JFrame implemented by XMLDiffFrame. Note that XMLDiffFrame instantiates the XMLSourceView bean, which is deprecated. The method follows these steps:

   a. Create the source pane for the input documents. Pass the DOM handles of the two documents to the diffFrame object to make the source pane:

   ```java
diffFrame.makeSrcPane(xmlDiff.getDocument1(), xmlDiff.getDocument2());
```

   b. Create the pane that shows the differences between the documents. Pass references to the text panes to diffFrame as follows:

   ```java
diffFrame.makeDiffSrcPane(new XMLDiffSrcView(xmlDiff.getDiffPane1()),
                        new XMLDiffSrcView(xmlDiff.getDiffPane2()));
```

   c. Create the pane for the XSLT stylesheet. Pass the DOM of the stylesheet as follows:

   ```java
diffFrame.makeXslPane(xslDoc, "Diff XSL Script");
diffFrame.makeXslTabbedPane();
```
This chapter contains these topics:

- Introduction to the XML SQL Utility (XSU)
- Using the XML SQL Utility: Overview
- Programming with the XSU Java API
- Programming with the XSU PL/SQL API
- Tips and Techniques for Programming with XSU

### Introduction to the XML SQL Utility (XSU)

XML SQL Utility (XSU) is an XDK component that enables you to transfer XML data through Oracle SQL statements. You can use XSU to perform the following tasks:

- Transform data in object-relational database tables or views into XML. XSU can query the database and return the result set as an XML document.
- Extract data from an XML document and use canonical mapping to insert the data into a table or a view or update or delete values of the appropriate columns or attributes.

This section contains the following topics:

- Prerequisites
- XSU Features
- XSU Restrictions

### Prerequisites

This chapter assumes that you are familiar with the following technologies:

- Oracle Database SQL. XSU transfers XML to and from a database through SELECT statements and DML.
- PL/SQL. The XDK supplies a PL/SQL API for XSU that mirrors the Java API.
- Java Database Connectivity (JDBC). Java applications that use XSU to transfer XML to and from a database require a JDBC connection.

### XSU Features

XSU has the following key features:

- Dynamically generates DTDs or XML schemas.
Using the XML SQL Utility: Overview

Generates XML documents in their string or DOM representations.

Performs simple transformations during generation such as modifying default tag names for each `<ROW>` element. You can also register an XSL transformation that XSU applies to the generated XML documents as needed.

Generates XML as a stream of SAX2 callbacks.

Supports XML attributes during generation, which enables you to specify that a particular column or group of columns maps to an XML attribute instead of an XML element.

Allows SQL to XML tag escaping. Sometimes column names are not valid XML tag names. To avoid this problem you can either alias all the column names or turn on tag escaping.

Supports `XMLType` columns in objects or tables.

Inserts XML into relational database tables or views. When given an XML document, XSU can also update or delete records from a database object.

**XSU Restrictions**

Note the following restrictions when using XSU:

- XSU can only store data in a single table. You can store XML across tables, however, by using the Oracle XSLT processor to transform a document into multiple documents and inserting them separately. You can also define views over multiple tables and perform insertions into the views. If a view is non-updatable (because of complex joins), then you can use `INSTEAD OF` triggers over the views to perform the inserts.

- You cannot use XSU to load XML data stored in attributes into a database schema, but you can use an XSLT transformation to change the attributes into elements.

- By default XSU is case sensitive. You can either use the correct case or specify that case should be ignored.

- XSU cannot generate a relational database schema from an input DTD.

- Inserting into `XMLType` tables using XSU is not supported. `XMLType` columns are supported.

**Using the XML SQL Utility: Overview**

This section contains the following topics:

- Using XSU: Basic Process
- Installing XSU
- Running the XSU Demo Programs
- Using the XSU Command-Line Utility

**Using XSU: Basic Process**

XSU is accessible through the following interfaces:

- The `OracleXMLQuery` and `OracleXMLSave` Java classes in the `oracle.xml.sql.query` package. Use the `OracleXMLQuery` class to generate XML from relational data and `OracleXMLSave` class to perform DML.
The PL/SQL packages `DBMS_XMLQuery` and `DBMS_XMLSave`, which mirror the Java classes.

You can write the following types of XSU applications:

- Java programs that run inside the database and access the internal XSU Java API
- Java programs that run on the client and access the client-side XSU Java API
- PL/SQL programs that access XSU through PL/SQL packages

Generating XML with the XSU Java API: Basic Process

The `OracleXMLQuery` class makes up the XML generation part of the XSU Java API. **Figure 11–1** illustrates the basic process for generating XML with XSU.

The basic steps in **Figure 11–1** are as follows:

1. Create a JDBC connection to the database. Normally, you establish a connection with the `DriverManager` class, which manages a set of JDBC drivers. After the JDBC drivers are loaded, call `getConnection()`. When it finds the right driver, this method returns a `Connection` object that represents a database session. All SQL statements are executed within the context of this session.

   You have the following options:
   
   - Create the connection with the JDBC OCI driver. The following code fragment illustrates this technique:
     ```java
     // import the Oracle driver class
     import oracle.jdbc.*;
     // load the Oracle JDBC driver
     DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
     // create the connection
     Connection conn =
     DriverManager.getConnection("jdbc:oracle:oci:@","hr","password");
     ``
   
   The preceding example uses the default connection for the JDBC OCI driver.
   
   - Create the connection with the JDBC thin driver. The thin driver is written in pure Java and can be called from any Java program. The following code fragment illustrates this technique:
     ```java
     Connection conn =
     DriverManager.getConnection("jdbc:oracle:thin:@dlsun489:1521:ORCL","hr","password");
     ``
   
   The thin driver requires the host name (dlsun489), port number (1521), and the Oracle SID (ORCL). The database must have an active TCP/IP listener.
Use default connection used by the server-side internal JDBC driver. This driver runs within a default session and default transaction context. You are already connected to the database; your SQL operations are part of the default transaction. Thus, you do not need to register the driver. Create the `Connection` object as follows:

```java
Connection conn = new oracle.jdbc.OracleDriver().defaultConnection();
```

**Note:** OracleXMLDataSetExtJdbc is used only for Oracle JDBC, whereas OracleXMLDataSetGenJdbc is used for non-Oracle JDBC. These classes are in the oracle.xml.sql.dataset package.

2. Create an XML query object and assign it a SQL query. You create an `OracleXMLQuery` class instance by passing a SQL query to the constructor, as shown in the following example:

```java
OracleXMLQuery qry = new OracleXMLQuery (conn, "SELECT * from EMPLOYEES");
```

3. Configure the XML query object by invoking `OracleXMLQuery` methods. The following example specifies that only 20 rows should be included in the result set:

```java
xmlQry.setMaxRows(20);
```

4. Return a DOM object or string by invoking `OracleXMLQuery` methods. For example, obtain a DOM object as follows:

```java
XMLDocument domDoc = (XMLDocument)qry.getXMLDOM();
```

Obtain a string object as follows:

```java
String xmlString = qry.getXMLString();
```

5. Perform additional processing on the string or DOM as needed.

**See Also:**

- Oracle Database Java Developer’s Guide to learn about Oracle JDBC
- Oracle Database XML Java API Reference to learn about OracleXMLQuery methods

**Performing DML with the XSU Java API: Basic Process**

Use the OracleXMLSave class to insert, update, and delete XML in the database. **Figure 11–2** illustrates the basic process.
The basic steps in Figure 11–2 are as follows:

1. Create a JDBC connection to the database. This step is identical to the first step described in "Generating XML with the XSU Java API: Basic Process" on page 11-3.

2. Create an XML save object and assign it a table on which to perform DML. Pass a table or view name to the constructor, as shown in the following example:

   ```java
   OracleXMLSave sav = new OracleXMLSave(conn, "employees");
   ```

3. Specify the primary key columns. For example, the following code specifies that `employee_id` is the key column:

   ```java
   String[] keyColNames = new String[1];
   keyColNames[0] = "EMPLOYEE_ID";
   sav.setKeyColumnList(keyColNames);
   ```

4. Configure the XML save object by invoking `OracleXMLSave` methods. The following example specifies an update of the `salary` and `job_id` columns:

   ```java
   String[] updateColNames = new String[2];
   updateColNames[0] = "SALARY";
   updateColNames[1] = "JOB_ID";
   sav.setUpdateColumnList(updateColNames); // set the columns to update
   ```

5. Invoke the `insertXML()`, `updateXML()`, or `deleteXML()` methods on the `OracleXMLSave` object. The following example illustrates an update:

   ```java
   // Assume that the user passes in this XML document as the first argument
   sav.updateXML(sav.getURL(argv[0]));
   ```

When performing the DML, XSU performs the following tasks:

- Parses the input XML document.
- Matches element names to column names in the target table or view.
c. Converts the elements to SQL types and binds them to the appropriate statement.

6. Close the OracleXMLSave object and deallocate all contexts associated with it, as shown in the following example:

```java
sav.close();
```

See Also:

- Oracle Database Java Developer’s Guide to learn about JDBC
- Oracle Database XML Java API Reference to learn about OracleXMLSave

Generating XML with the XSU PL/SQL API: Basic Process

The XSU PL/SQL API reflects the Java API in the generation and storage of XML documents from and to a database. DBMS_XMLQuery is the PL/SQL package that reflects the methods in the OracleXMLQuery Java class. This package has a context handle associated with it. Create a context by calling one of the constructor-like functions to get the handle and then use the handle in all subsequent calls.

---

**Note:** For improved performance, consider using the C-based DBMS_XMLGEN, which is written in C and built into the database, rather than DBMS_XMLQUERY.

---

XSU supports the XMLType datatype. Using XSU with XMLType is useful if, for example, you have XMLType columns in objects or tables.

Generating XML results in a CLOB that contains the XML document. To use DBMS_XMLQuery and the XSU generation engine, follow these basic steps:

1. Declare a variable for the XML query context and a variable for the generated XML. For example:

```java
v_queryCtx  DBMS_XMLQuery.ctxType;
v_result    CLOB;
```

2. Obtain a context handle by calling the DBMS_XMLQuery.newContext function and supplying it the query, either as a CLOB or a VARCHAR2. The following example registers a query to select the rows from the employees table with the WHERE clause containing the bind variables :EMPLOYEE_ID and :FIRST_NAME:

```java
v_queryCtx = DBMS_XMLQuery.newContext('SELECT * FROM employees
WHERE employee_id=:EMPLOYEE_ID AND first_name=:FIRST_NAME');
```

3. Bind values to the query. The binds work by binding a name to the position. clearBindValues clears all the bind variables, whereas setBindValue sets a single bind variable with a string value. For example, bind the employee_id and first_name values as shown:

```java
DBMS_XMLQuery.setBindValue(v_queryCtx, 'EMPLOYEE_ID', 20);
DBMS_XMLQuery.setBindValue(v_queryCtx, 'FIRST_NAME', 'John');
```

4. Configure the query context. Set optional arguments such as the ROW tag name, the ROWSET tag name, or the number of rows to fetch, and so on. The following example specifies changes the default ROWSET element name to EMPSET:

```java
DBMS_XMLQuery.setRowSetTag(v_queryCtx, 'EMPSET');
```
5. Fetch the results. You can obtain the XML as a CLOB with the `getXML` function, which generates XML with or without a DTD or XML schema. The following example applies bind values to the statement and gets the result corresponding to the predicate `employee_id = 20` and `first_name = 'John'`:

```sql
v_result := DBMS_XMLQuery.getXML(v_queryCtx);
```

6. Process the results of the XML generation. For example, suppose that your program declared the following variables:

```sql
v_xmlstr VARCHAR2(32767);
v_line   VARCHAR2(2000);
```

You can print the CLOB stored in `v_result` as follows:

```sql
v_xmlstr := DBMS_LOB.SUBSTR(v_result,32767);
LOOP
  EXIT WHEN v_xmlstr IS NULL;
  v_line := substr(v_xmlstr,1,INSTR(v_xmlstr,CHR(10))-1);
  DBMS_OUTPUT.PUT_LINE('| ' || v_line);
  v_xmlstr := SUBSTR(v_xmlstr,INSTR(v_xmlstr,CHR(10))+1);
END LOOP;
```

7. Close the context. For example:

```sql
DBMS_XMLQuery.closeContext(v_queryCtx);
```

### Performing DML with the PL/SQL API: Basic Process

`DBMS_XMLSave` is the PL/SQL package that reflects the methods in the OracleXMLSave Java class. This package has a context handle associated with it. Create a context by calling one of the constructor-like functions to get the handle and then use the handle in all subsequent calls.

To use `DBMS_XMLSave`, follow these basic steps:

1. Declare a variable for the XML save context and a variable for the number of rows touched by the DML. For example:

   ```sql
   savCtx DBMS_XMLSave.ctxType;
v_rows   NUMBER;
   ```

2. Create a context handle by calling the `DBMS.XMLSave.newContext` function and supply it the table name to use for the DML operations.

   ```sql
   savCtx  := DBMS.XMLSave.newContext('hr.employees');
   ```

3. Set options based on the type of DML that you want to perform.

   For inserts you can set the list of columns to insert into the `setUpdateColumn` function. The default is to insert values into all columns. The following example sets five columns in the `employees` table:

   ```sql
   DBMS_XMLSave.setUpdateColumn(savCtx,'EMPLOYEE_ID');
   DBMS_XMLSave.setUpdateColumn(savCtx,'LAST_NAME');
   DBMS_XMLSave.setUpdateColumn(savCtx,'EMAIL');
   DBMS_XMLSave.setUpdateColumn(savCtx,'JOB_ID');
   DBMS_XMLSave.setUpdateColumn(savCtx,'HIRE_DATE');
   ```

   For updates you must supply the list of key columns. Optionally, you can then supply the list of columns for update. In this case, the tags in the XML document matching the key column names will be used in the `WHERE` clause of the `UPDATE`
statement and the tags matching the update column list will be used in the SET clause of the UPDATE statement. For example:

```sql
DBMS_XMLSave.setKeyColumn(savCtx,'employee_id'); -- set key column
-- set list of columns to update.
DBMS_XMLSave.updateColumn(savCtx,'salary');
DBMS_XMLSave.updateColumn(savCtx,'job_id');
```

For deletes the default is to create a WHERE clause to match all the tag values present in each <ROW> element of the document supplied. To override this behavior, set the list of key columns. In this case only those tag values whose tag names match these columns are used to identify the rows to delete (in effect used in the WHERE clause of the DELETE statement). For example:

```sql
DBMS_XMLSave.setKeyColumn(savCtx,'EMPLOYEE_ID');
```

4. Supply a context and XML document to the insertXML, updateXML, or deleteXML functions. For example:

```sql
v_rows := DBMS_XMLSave.deleteXML(savCtx,xmlDoc);
```

5. Repeat the DML any number of times if needed.

6. Close the context. For example:

```sql
DBMS_XMLSave.closeContext(savCtx);
```

For a model use the Java examples described in "Programming with the XSU Java API" on page 11-17.

**Installing XSU**

XSU is included in the Oracle Database software CD along with the other XDK utilities. "Java XDK Component Dependencies" on page 3-2 describes the XSU components and dependencies.

By default, the Oracle Universal Installer installs XSU on disk and loads it into the database. No user intervention is required. If you did not load XSU in the database when installing Oracle, you can install XSU manually as follows:

1. Make sure that Oracle XML DB is installed.
2. Load the xsu12.jar file into the database. This JAR file, which has a dependency on xdb.jar for XMLType access, is described in Table 3–1 on page 3-3.
3. Run the `$ORACLE_HOME/rdbms/admin/dbmsxsu.sql` script. This SQL script builds the XSU PL/SQL API.

As explained in "Using XSU: Basic Process" on page 11-2, you do not have to load XSU into the database in order to use it. XSU can reside in any tier that supports Java.

The following sections describe your installation options:

- Installing XSU in the Database
- Installing XSU in an Application Server
- Installing XSU in a Web Server
Installing XSU in the Database

Figure 11–3 shows the typical architecture for applications that use the XSU libraries installed in the database. XML generated from XSU running in the database can be placed in advanced queues in the database to be queued to other systems or clients. You deliver the XML internally through stored procedures in the database or externally through web or application servers.

In Figure 11–3 all lines are bidirectional. Because XSU can generate as well as save data, resources can deliver XML to XSU running inside the database, which can then insert it in the appropriate database tables.

Installing XSU in an Application Server

Your application architecture may need to use an application server in the middle tier. The application tier can be an Oracle database, Oracle application server, or a third-party application server that supports Java programs.

You can generate XML in the middle tier from SQL queries or ResultSets for various reasons, for example, to integrate different JDBC data sources in the middle tier. In this case, you can install the XSU in your middle tier, thereby enabling your Java programs to make use of XSU through its Java API.

Figure 11–4 shows a typical architecture for running XSU in a middle tier. In the middle tier, data from JDBC sources is converted by XSU into XML and then sent to Web servers or other systems. Again, the process is bidirectional, which means that the data can be put back into the JDBC sources (database tables or views) by means of XSU. If an Oracle database itself is used as the application server, then you can use the PL/SQL front-end instead of Java.
Installing XSU in a Web Server

Figure 11–5 shows that XSU can live in the Web server as long as the Web server supports Java servlets. In this way you can write Java servlets that use XSU. XSQL Servlet is a standard servlet provided by Oracle. It is built on top of XSU and provides a template-like interface to XSU functionality. To perform XML processing in the Web server and avoid intricate servlet programming, you can use the XSQL Servlet.

Figure 11–5 Running XSU in a Web Server
See Also:

- *Oracle XML DB Developer’s Guide*, especially the chapter on generating XML, for examples on using XSU with XMLType
- *Oracle Database XML Java API Reference* to learn about the classes OracleXMLQuery and OracleXMLSave
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the packages DBMS_XMLQuery and DBMS_XMLSave
- Chapter 14, "Using the XSQL Pages Publishing Framework" to learn about XSQL Servlet

Running the XSU Demo Programs

Demo programs for XSU are included in $ORACLE_HOME/xdk/demo/java/xsu. Table 11–1 describes the XML files and programs that you can use to test XSU.

<table>
<thead>
<tr>
<th>Table 11–1 XSU Sample Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bindSQLVariables.sql</td>
<td>An PL/SQL script that binds values for EMPLOYEE_ID and FIRST_NAME to columns in the employees table. Refer to “Binding Values in XSU” on page 11-31.</td>
</tr>
<tr>
<td>changeElementName.sql</td>
<td>A PL/SQL program that obtains the first 20 rows of the employees table as an XML document. Refer to “Specifying Element Names with DBMS_XMLQuery” on page 11-30.</td>
</tr>
<tr>
<td>createObjRelSchema.sql</td>
<td>A SQL script that sets up an object-relational schema and populates it. Refer to “XML Mapping Against an Object-Relational Schema” on page 11-38.</td>
</tr>
<tr>
<td>createObjRelSchema2.sql</td>
<td>A SQL script that sets up an object-relational schema and populates it. Refer to “Altering the Database Schema or SQL Query” on page 11-40.</td>
</tr>
<tr>
<td>createRelSchema.sql</td>
<td>A SQL script that creates a relational table and then creates a customer view that contains a customer object on top of it. Refer to “Altering the Database Schema or SQL Query” on page 11-40.</td>
</tr>
<tr>
<td>customer.xml</td>
<td>An XML document that describes a customer. Refer to “Altering the Database Schema or SQL Query” on page 11-40.</td>
</tr>
<tr>
<td>deleteEmployeeByKey.sql</td>
<td>A PL/SQL program that deletes an employee by primary key. Refer to “Deleting by Key with DBMS_XMLSave: Example” on page 11-36.</td>
</tr>
<tr>
<td>deleteEmployeeByRow.sql</td>
<td>A PL/SQL program that deletes an employee by row. Refer to “Deleting by Row with DBMS_XMLSave: Example” on page 11-35.</td>
</tr>
<tr>
<td>domTest.java</td>
<td>A program that generates a DOM tree and then traverses it in document order, printing the nodes one by one. Refer to “Generating a DOM Tree with OracleXMLQuery” on page 11-18.</td>
</tr>
<tr>
<td>index.txt</td>
<td>A README that describes the programs in the demo directory.</td>
</tr>
<tr>
<td>insProc.sql</td>
<td>A PL/SQL program that inserts an XML document into a table. Refer to “Inserting Values into All Columns with DBMS_XMLSave” on page 11-32.</td>
</tr>
<tr>
<td>insertClob.sql</td>
<td>A SQL script that creates a table called xmldocument and stores an XML document in the table as a CLOB. Refer to “Inserting Values into All Columns with DBMS_XMLSave” on page 11-32.</td>
</tr>
<tr>
<td>insertClob2.sql</td>
<td>A SQL script that inserts an XML document into the xmldocument table. Refer to “Inserting into a Subset of Columns with DBMS_XMLSave” on page 11-33.</td>
</tr>
<tr>
<td>insertClob3.sql</td>
<td>A SQL script that inserts an XML document into the xmldocument table. Refer to “Updating with Key Columns with DBMS_XMLSave” on page 11-34.</td>
</tr>
<tr>
<td>insertClob4.sql</td>
<td>A SQL script that inserts an XML document into the xmldocument table. Refer to “Specifying a List of Columns with DBMS_XMLSave: Example” on page 11-35.</td>
</tr>
<tr>
<td>insertEmployee.sql</td>
<td>A PL/SQL script that calls the insProc stored procedure and inserts an employee into the employees table. Refer to “Inserting XML with DBMS_XMLSave” on page 11-31.</td>
</tr>
<tr>
<td>insertEmployee2.sql</td>
<td>A PL/SQL script that invokes the testInsert procedure to insert the XML data for an employee into the hr.employees table. Refer to “Inserting into a Subset of Columns with DBMS_XMLSave” on page 11-33.</td>
</tr>
</tbody>
</table>
Table 11–1 (Cont.) XSU Sample Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapColumnToAtt.sql</td>
<td>A SQL script that queries the employees table, rendering employee_id as an XML attribute. Refer to “Altering the Database Schema or SQL Query” on page 11-40.</td>
</tr>
<tr>
<td>new_emp.xml</td>
<td>An XML document that describes a new employee. Refer to “Running the testInsert Program” on page 11-23.</td>
</tr>
<tr>
<td>new_emp2.xml</td>
<td>An XML document that describes a new employee. Refer to “Running the testInsertSubset Program” on page 11-24.</td>
</tr>
<tr>
<td>noRowsTest.java</td>
<td>A program that throws an exception when there are no more rows. Refer to “Raising a No Rows Exception” on page 11-29.</td>
</tr>
<tr>
<td>pageTest.java</td>
<td>A program that uses the JDBC ResultSet to generate XML one page at a time. Refer to “Generating Scrollable Result Sets” on page 11-21.</td>
</tr>
<tr>
<td>paginateResults.java</td>
<td>A program that generates an XML page that paginates results. Refer to “Generating Results with OracleXMLQuery: Example” on page 11-20.</td>
</tr>
<tr>
<td>paginateResults.sql</td>
<td>A PL/SQL script that paginates results. It skips the first 3 rows of the employees table and then prints the rest of the rows 10 at a time by setting skipRows to 3 for the first batch of 10 rows and then to 0 for the rest of the batches. Refer to “Paginating Results with DBMS_XMLQuery” on page 11-31.</td>
</tr>
<tr>
<td>printClobOut.sql</td>
<td>A PL/SQL script that prints a CLOB to the output buffer. Refer to “Generating XML from Simple Queries with DBMS_XMLQuery” on page 11-30.</td>
</tr>
<tr>
<td>raiseException.sql</td>
<td>A PL/SQL script that invokes the DBMS_XMLQuery.getExceptionContent procedure. Refer to “Handling Exceptions in the XSU PL/SQL API” on page 11-36.</td>
</tr>
<tr>
<td>refCurTest.java</td>
<td>A program that generates XML from the results of the SQL query defined in the testRefCur function. Refer to “Generating XML from Cursor Objects” on page 11-22.</td>
</tr>
<tr>
<td>samp1.java</td>
<td>A program that queries the scott.emp table, then generates an XML document from the query results.</td>
</tr>
<tr>
<td>samp10.java</td>
<td>A program that inserts sampdoc.xml into the xmltest_tab1 table.</td>
</tr>
<tr>
<td>samp2.java</td>
<td>A program that queries the scott.emp table, then generates an XML document from the query results. This program demonstrates how you can customize the generated XML document.</td>
</tr>
<tr>
<td>sampdoc.xml</td>
<td>A sample XML data document that samp10.java inserts into the database.</td>
</tr>
<tr>
<td>samps.sql</td>
<td>A SQL script that creates the xmltest_tab1 table used by samp10.java.</td>
</tr>
<tr>
<td>simpleQuery.sql</td>
<td>A PL/SQL script that selects 20 rows from the hr.employees table and obtains an XML document as a CLOB. Refer to “Generating XML from Simple Queries with DBMS_XMLQuery” on page 11-30.</td>
</tr>
<tr>
<td>testDML.sql</td>
<td>A PL/SQL script that uses the same context and settings to perform DML depending on user input. Refer to “Reusing the Context Handle with DBMS_XMLSave” on page 11-36.</td>
</tr>
<tr>
<td>testDeleteKey.java</td>
<td>A program that limits the number of elements used to identify a row, which improves performance by caching the DELETE statement and batching transactions. Refer to “Deleting by Key with OracleXMLSave” on page 11-28.</td>
</tr>
<tr>
<td>testDeleteKey.sql</td>
<td>A PL/SQL script that deletes a row from the employees table for every &lt;ROW&gt; element in an input XML document. Refer to “Deleting by Key with DBMS_XMLSave: Example” on page 11-35.</td>
</tr>
<tr>
<td>testDeleteRow.java</td>
<td>A program that accepts an XML document filename as input and deletes the rows corresponding to the elements in the document. Refer to “Deleting by Row with OracleXMLSave” on page 11-27.</td>
</tr>
<tr>
<td>testDeleteRow.sql</td>
<td>A SQL script that deletes a row from the employees table for every &lt;ROW&gt; element in an input XML document. Refer to “Deleting by Row with DBMS_XMLSave: Example” on page 11-35.</td>
</tr>
<tr>
<td>testException.java</td>
<td>A sample program shown that throws a runtime exception and then obtains the parent exception by invoking Exception.getException(). Refer to “Obtaining the Parent Exception” on page 11-29.</td>
</tr>
<tr>
<td>testInsert.java</td>
<td>A Java program that inserts XML values into all columns of the hr.employees table. Refer to “Inserting XML into All Columns with OracleXMLSave” on page 11-22.</td>
</tr>
<tr>
<td>testInsert.sql</td>
<td>A PL/SQL script that inserts XML data into a subset of columns. Refer to “Inserting into a Subset of Columns with DBMS_XMLSave” on page 11-33.</td>
</tr>
</tbody>
</table>
The steps for running the demos are:

1. Change into the `$ORACLE_HOME/xdk/demo/java/xsu` directory (UNIX) or `%ORACLE_HOME%\xdk\demo\java\xsu` directory (Windows).

2. Make sure that your environment variables are set as described in "Setting Up the Java XDK Environment" on page 3-5. In particular, make sure that the Java classpath includes `xsu12.jar` for XSU and `ojdbc5.jar` (Java 1.5) for JDBC. If you use a multibyte character set other than UTF-8, ISO8859-1, or JA16SJIS, then place `orai18n.jar` in your classpath so that JDBC can convert the character set of the input file to the database character set.

3. Compile the Java programs as shown in the following example:
   ```
   javac samp1.java samp2.java samp10.java
   ```

4. Connect to an Oracle database as `hr` and run the SQL script `createRelSchema`:
   ```
   CONNECT hr
   @$ORACLE_HOME/xdk/demo/java/xsu/createRelSchema
   ```

The following sections describe the XSU demos in detail.
Using the XSU Command-Line Utility

The XDK includes a command-line Java interface for XSU. XSU command-line options are provided through the Java class OracleXML. To use this API ensure that your Java classpath is set as described in "Setting Up the Java XDK Environment" on page 3-5.

To print usage information for XSU to standard output, run the following command:

```
java OracleXML
```

To use XSU, invoke it with either the `getXML` or `putXML` parameter as follows:

```java
java OracleXML getXML options
java OracleXML putXML options
```

Table 11–2 describes the `getXML` options.

<table>
<thead>
<tr>
<th>getXML Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-user &quot;username/password&quot;</code></td>
<td>Specifies the username and password to connect to the database. If this is not specified, then the user defaults to <code>scott/tiger</code>. Note that the connect string is also specified. You can specify the username and password as part of the connect string.</td>
</tr>
<tr>
<td><code>-conn &quot;JDBC_connect_string&quot;</code></td>
<td>Specifies the JDBC database connect string. By default the connect string is: <code>jdbc:oracle:oci:@</code>.</td>
</tr>
<tr>
<td><code>-withDTD</code></td>
<td>Instructs the XSU to generate the DTD along with the XML document.</td>
</tr>
<tr>
<td><code>-withSchema</code></td>
<td>Instructs the XSU to generate the schema along with the XML document.</td>
</tr>
<tr>
<td><code>-rowsetTag tag_name</code></td>
<td>Specifies the rowset tag, which is tag that encloses all the XML elements corresponding to the records returned by the query. The default rowset tag is <code>&lt;ROWSET&gt;</code>. If you specify an empty string (&quot;&quot;) for rowset, then XSU omits the rowset element.</td>
</tr>
<tr>
<td><code>-rowTag tag_name</code></td>
<td>Specifies the row tag that encloses the data corresponding to a database row. The default row tag is <code>&lt;ROW&gt;</code>. If you specify an empty string (&quot;&quot;) for the row tag, then XSU omits the row tag.</td>
</tr>
<tr>
<td><code>-rowIdAttr row_id_attribute_name</code></td>
<td>Names the attribute of the <code>ROW</code> element that keeps track of the cardinality of the rows. By default this attribute is <code>num</code>. If you specify an empty string (&quot;&quot;) as the <code>rowID</code> attribute, then XSU omits the attribute.</td>
</tr>
<tr>
<td><code>-rowIdColumn row_Id_column_name</code></td>
<td>Specifies that the value of one of the scalar columns from the query is to be used as the value of the <code>rowID</code> attribute.</td>
</tr>
<tr>
<td><code>-collectionIdAttr collect_id_attr_name</code></td>
<td>Names the attribute of an XML list element that keeps track of the cardinality of the elements of the list. The generated XML lists correspond to either a cursor query, or collection. If you specify an empty string (&quot;&quot;) as the <code>rowID</code> attribute, then XSU omits the attribute.</td>
</tr>
<tr>
<td><code>-useTypeForCollElemTag</code></td>
<td>Specifies the use type name for the column-element tag. By default XSU uses the <code>column-name_item</code>.</td>
</tr>
<tr>
<td><code>-useNullAttrId</code></td>
<td>Specifies the attribute <code>NULL</code> (TRUE/FALSE) to indicate the nullness of an element.</td>
</tr>
<tr>
<td><code>-stylesheet stylesheet_URI</code></td>
<td>Specifies the stylesheet in the XML processing instruction.</td>
</tr>
<tr>
<td><code>-stylesheetType stylesheet_type</code></td>
<td>Specifies the stylesheet type in the XML processing instruction.</td>
</tr>
<tr>
<td><code>-setXSLT URI</code></td>
<td>Specifies the XSLT stylesheet to apply to the XML document.</td>
</tr>
<tr>
<td><code>-setXSLTRef URI</code></td>
<td>Sets the XSLT external entity reference.</td>
</tr>
</tbody>
</table>
Table 11–3 describes the putXML options.

<table>
<thead>
<tr>
<th>putXML Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-user &quot;username/password&quot;</td>
<td>Specifies the username and password to connect to the database. If not specified, the user defaults to scott/tiger. The connect string is also specified; you can specify the username and password as part of the connect string.</td>
</tr>
<tr>
<td>-conn &quot;JDBC_connect_string&quot;</td>
<td>Specifies the JDBC database connect string. By default the connect string is: &quot;jdbc:oracle:oci:@&quot;.</td>
</tr>
<tr>
<td>-batchSize batching_size</td>
<td>Specifies the batch size that controls the number of rows that are batched together and inserted in a single trip to the database to improve performance.</td>
</tr>
<tr>
<td>-commitBatch commit_size</td>
<td>Specifies the number of inserted records after which a commit is to be executed. If the autocommit is TRUE (the default), then setting commitBatch has no consequence.</td>
</tr>
<tr>
<td>-rowTag tag_name</td>
<td>Specifies the row tag, which is tag used to enclose the data corresponding to a database row. The default row tag is &lt;ROW&gt;. If you specify an empty string for the row tag, then XSU omits the row tag.</td>
</tr>
<tr>
<td>-dateFormat date_format</td>
<td>Specifies the date format for the date values in the XML document.</td>
</tr>
<tr>
<td>-withEscaping</td>
<td>Turns on reverse mapping if SQL to XML name escaping was used when generating the doc.</td>
</tr>
<tr>
<td>-ignoreCase</td>
<td>Makes the matching of the column names with tag names case insensitive. For example, EmpNo matches with EMPNO if ignoreCase is on.</td>
</tr>
<tr>
<td>-preserveWhitespace</td>
<td>Preserves the whitespace in the inserted XML document.</td>
</tr>
<tr>
<td>-setXSLT URI</td>
<td>Specifies the XSLT to apply to the XML document before inserting.</td>
</tr>
</tbody>
</table>

Table 11–3 putXML Options
Generating XML with the XSU Command-Line Utility

To generate XML from the database schema use the `getXML` parameter. For example, to generate an XML document by querying the `employees` table in the `hr` schema, you can use the following syntax:

```java
java OracleXML getXML -user "hr/password" "SELECT * FROM employees"
```

The preceding command performs the following tasks:
1. Connects to the current default database
2. Executes the specified `SELECT` query
3. Converts the SQL result set to XML
4. Prints the XML to standard output

The `getXML` parameter supports a wide range of options, which are explained in Table 11–2.

Generating XMLType Data with the XSU Command-Line Utility

You can use XSU to generate XML from tables with XMLType columns. Suppose that you run the demo script `setup_xmltype.sql` to create and populate the `parts` table. You can generate XML from this table with XSU as follows:

```java
java OracleXML getXML -user "hr/password" -rowTag "Part" "SELECT * FROM parts"
```

The output of the command is shown below:

```xml
<?xml version = '1.0'?>
<ROWSET>
  <Part num="1">
    <PARTNO>1735</PARTNO>
    <PARTNAME>Gizmo</PARTNAME>
    <PARTDESC>
      <Description>
        <Title>Description of the Gizmo</Title>
        <Author>John Smith</Author>
        <Body>
          The <b>Gizmo</b> is <i>grand</i>.
        </Body>
      </Description>
    </PARTDESC>
  </Part>
</ROWSET>
```

Performing DML with the XSU Command-Line Utility

To insert an XML document called `new_employees.xml` into the `hr.employees` table, use the following syntax:

```java
java OracleXML putXML -user "hr/password" -fileName "new_employees.xml" employees
```

<table>
<thead>
<tr>
<th>putXML Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-setXSLTRef URI</td>
<td>Sets the XSLT external entity reference.</td>
</tr>
<tr>
<td>-fileName file_name</td>
<td>-URL URL</td>
</tr>
<tr>
<td>table_name</td>
<td>Specifies the name of the table to put the values into.</td>
</tr>
</tbody>
</table>
The preceding command performs the following tasks:
1. Connects to the current database as hr
2. Reads the XML document named new_emp.xml
3. Parses the XML document, matching the tags with column names
4. Inserts the values appropriately into the employees table

The `getXML` parameter supports a wide range of options, which are explained in Table 11–2.

**Programming with the XSU Java API**

This section contains the following topics:
- Generating a String with OracleXMLQuery
- Generating a DOM Tree with OracleXMLQuery
- Paginating Results with OracleXMLQuery
- Generating Scrollable Result Sets
- Generating XML from Cursor Objects
- Inserting Rows with OracleXMLSave
- Updating Rows with OracleXMLSave
- Deleting Rows with OracleXMLSave
- Handling XSU Java Exceptions

**Generating a String with OracleXMLQuery**

The `testXMLSQL.java` demo program uses XSU to generate XML as a `String` object. This program queries the `hr.employees` table and prints the result set to standard output.

The `testXMLSQL.java` program follows these steps:
1. Register the JDBC driver and create a database connection. The following code fragment uses the OCI JDBC driver and connects with the username hr:
   ```java
   import oracle.jdbc.*;
   ...
   Connection conn  = getConnection("hr","password");
   ...
   private static Connection getConnection(String username, String password)
   throws SQLException
   {
     // register the JDBC driver
     DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
     // create the connection using the OCI driver
     Connection conn =
       DriverManager.getConnection("jdbc:oracle:oci:@",username,password);
     return conn;
   }
   ``

2. Create an XML query object and initialize it with a SQL query. The following code fragment initializes the object with a `SELECT` statement on `hr.employees`:
OracleXMLQuery qry = new OracleXMLQuery(conn, "SELECT * FROM employees");

3. Obtain the query result set as a String object. The getXMLString() method transforms the object-relational data specified in the constructor into an XML document. The following example illustrates this technique:

   String str = qry.getXMLString();

4. Close the query object to release any resources, as shown in the following code:

   qry.close();

Running the testXMLSQL Program

To run the testXMLSQL.java program perform the following steps:

1. Compile testXMLSQL.java with javac.

2. Execute java testXMLSQL on the command line.

You must have the CLASSPATH pointing to this directory for the Java executable to find the class. Alternatively, use visual Java tools such as Oracle JDeveloper to compile and run this program. When run, this program prints out the XML file to the screen. The following shows sample output with some rows edited out:

   <?xml version = '1.0'?>
   <ROWSET>
     <ROW num="1">
       <EMPLOYEE_ID>100</EMPLOYEE_ID>
       <FIRST_NAME>Steven</FIRST_NAME>
       <LAST_NAME>King</LAST_NAME>
       <EMAIL>SKING</EMAIL>
       <PHONE_NUMBER>515.123.4567</PHONE_NUMBER>
       <HIRE_DATE>6/17/1987 0:0:0</HIRE_DATE>
       <JOB_ID>AD_PRES</JOB_ID>
       <SALARY>24000</SALARY>
       <DEPARTMENT_ID>90</DEPARTMENT_ID>
     </ROW>
   <!-- ROW num="2" through num="107" ... -->
   </ROWSET>

Generating a DOM Tree with OracleXMLQuery

To generate a DOM tree from the XML generated by XSU, you can directly request a DOM document from XSU. This technique saves the overhead of creating a string representation of the XML document and then parsing it to generate the DOM tree.

XSU calls the Oracle XML parser to construct the DOM tree from the data values. The domTest.java demo program generates a DOM tree and then traverses it in document order, printing the nodes one by one.

The first two steps in the domTest.java program are the same as for the testXMLSQL.java program described in "Generating a String with OracleXMLQuery" on page 11-17. The program proceeds as follows:

1. Obtain the DOM by invoking getXMLDOM() method. The following example illustrates this technique:

   XMLDocument domDoc = (XMLDocument)qry.getXMLDOM();

2. Print the DOM tree. The following code prints to standard output:

   domDoc.print(System.out);
You can also create a `StringWriter` and wrap it in a `PrintWriter` as follows:

```java
StringWriter s = new StringWriter(10000);
domDoc.print(new PrintWriter(s));
System.out.println("The string version ---> 
"+s.toString());
```

After compiling the program, run it from the command line as follows:

```bash
java domTest
```

### Paginating Results with OracleXMLQuery

This section contains the following topics:

- Limiting the Number of Rows in the Result Set
- Keeping the Object Open for the Duration of the User's Session
- Paginating Results with OracleXMLQuery: Example

#### Limiting the Number of Rows in the Result Set

In `testXMLSQL.java` and `domTest.java`, XSU generated XML from all rows returned by the query. Suppose that you query a table that contains 1000 rows, but you want only 100 rows at a time. One approach is to execute one query to obtain the first 100 rows, another to obtain the next 100 rows, and so on. With this technique you cannot skip the first five rows of the query and then generate the result. To avoid these problems, use the following Java methods:

- `OracleXMLSave.setSkipRows()` forces XSU to skip the desired number of rows before starting to generate the result. The command-line equivalent to this method is the `-skipRows` parameter.
- `OracleXMLSave.setMaxRows()` limits the number of rows converted to XML. The command-line equivalent to this method is the `-maxRows` parameter.

**Example 11–1** sets `skipRows` to a value of 5 and `maxRows` to a value of 1, which causes XSU to skip the first 5 rows and then generate XML for the next row when querying the `hr.employees` table.

```java
Example 11–1 Specifying skipRows and maxRows on the Command Line
java OracleXML getXML -user "hr/password" -skipRows 5 -maxRows 1 "SELECT * FROM employees"
```

The following shows sample output (only row 6 of the query result set is returned):

```xml
<?xml version = '1.0'?><ROWSET>
  <ROW num="6">
    <EMPLOYEE_ID>105</EMPLOYEE_ID>
    <FIRST_NAME>David</FIRST_NAME>
    <LAST_NAME>Austin</LAST_NAME>
    <EMAIL>DAUSTIN</EMAIL>
    <PHONE_NUMBER>590.423.4569</PHONE_NUMBER>
    <HIRE_DATE>6/25/1997 0:0:0</HIRE_DATE>
    <JOB_ID>IT_PROG</JOB_ID>
    <SALARY>4800</SALARY>
    <MANAGER_ID>103</MANAGER_ID>
    <DEPARTMENT_ID>60</DEPARTMENT_ID>
  </ROW>
</ROWSET>
```
Keeping the Object Open for the Duration of the User's Session

In some situations you may want to keep the query object open for the duration of the user session. You can handle such cases with the maxRows() method and the keepObjectOpen() method.

Consider a Web search engine that paginates search results. The first page lists 10 results, the next page lists 10 more, and so on. To perform this task with XSU, request 10 rows at a time and keep the ResultSet open so that the next time you ask XSU for more results, it starts generating from where the last generation finished. If OracleXMLQuery creates a result set from the SQL query string, then it typically closes the ResultSet internally because it assumes no more results are required. Thus, you should invoke keepObjectOpen() to keep the cursor active.

A different case requiring an open query object is when the number of rows or number of columns in a row is very large. In this case, you can generate multiple small documents rather than one large document.

See Also: "Paginating Results with OracleXMLQuery: Example" on page 11-20

Paginating Results with OracleXMLQuery: Example

The paginateResults.java program shows how you can generate an XML page that paginates results. The output XML displays only 20 rows of the hr table.

The first step of the paginateResults.java program, which creates the connection, is the same as in testXMLSQL.java. The program continues as follows:

1. Create a SQL statement object and initialize it with a SQL query. The following code fragment sets two options in java.sql.ResultSet:
   ```java
   Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
   ResultSet.CONCUR_READ_ONLY);
   ```

2. Create the query as a string and execute it by invoking Statement.executeQuery(). The return object is of type ResultSet. The following example illustrates this technique:
   ```java
   String sCmd = "SELECT first_name, last_name FROM hr.employees";
   ResultSet rs = stmt.executeQuery(sCmd);
   ```

3. Create the query object, as shown in the following code:
   ```java
   OracleXMLQuery xmlQry = new OracleXMLQuery(conn, rs);
   ```

4. Configure the query object. The following code specifies that the query object should be open for the duration of the session. It also limits the number of rows returned to 20:
   ```java
   xmlQry.keepObjectOpen(true);
   xmlQry.setRowsetTag("ROWSET");
   xmlQry.setRowTag("ROW");
   xmlQry.setMaxRows(20);
   ```

5. Retrieve the result as a String and print:
   ```java
   String sXML = xmlQry.getXMLString();
   System.out.println(sXML);
   ```

After compiling the program, run it from the command line as follows:

```
java paginateResults
```
Generating Scrollable Result Sets

In some situations you may want to perform a query and then retrieve a previous page of results from within the result set. To enable scrolling, instantiate the Oracle.jdbc.ResultSet class. You can use the ResultSet object to move back and forth within the result set and use XSU to generate XML each time.

The pageTest.java program shows how to use the JDBC ResultSet to generate XML a page at a time. Using ResultSet may be necessary in cases that are not handled directly by XSU, for example, when setting the batch size and binding values.

The pageTest.java program creates a pageTest object and initializes it with a SQL query. The constructor for the pageTest object performs the following steps:

1. Create a JDBC connection by calling the same getConnection() method defined in paginateResults.java:
   ```java
   Connection conn;
   ...
   conn = getConnection("hr", "password");
   ```

2. Create a statement as follows:
   ```java
   Statement stmt;
   ...
   stmt = conn.createStatement();
   ```

3. Execute the query passed to the constructor to obtain the scrollable result set. The following code illustrates this technique:
   ```java
   ResultSet rset = stmt.executeQuery(sqlQuery);
   ```

4. Create a query object by passing references to the connection and result set objects to the constructor. The following code fragment illustrates this technique:
   ```java
   OracleXMLQuery qry;
   ...
   qry = new OracleXMLQuery(conn, rset);
   ```

5. Configure the query object. The following code fragment specifies that the query object should be kept open and that it should raise an exception when there are no more rows:
   ```java
   qry.keepObjectOpen(true);
   qry.setRaiseNoRowsException(true);
   qry.setRaiseException(true);
   ```

6. After creating the query object by passing it the string "SELECT * FROM employees", the program loops through the result set. The getResult() method receives integer values specifying the start row and end row of the set. It sets the maximum number of rows to retrieve by calculating the difference of these values and then retrieves the result as a string. The following while loop retrieves and prints ten rows at a time:
   ```java
   int i = 0;
   while ((str = test.getResult(i, i+10))!= null)
   {
       System.out.println(str);
       i+= 10;
   }
   ```

After compiling the program, run it from the command line as follows:
Generating XML from Cursor Objects

The OracleXMLQuery class provides XML conversion only for query strings or ResultSet objects. If your program uses PL/SQL procedures that return REF cursors, then how do you perform the conversion? You can use the ResultSet conversion mechanism described in "Generating Scrollable Result Sets" on page 11-21.

REF cursors are references to cursor objects in PL/SQL. These cursor objects are SQL statements over which a program can iterate to obtain a set of values. The cursor objects are converted into OracleResultSet objects in the Java world. In your Java program you can initialize a CallableStatement object, execute a PL/SQL function that returns a cursor variable, obtain the OracleResultSet object, and then send it to the OracleXMLQuery object to obtain the desired XML.

Consider the testRef PL/SQL package defined in the testRef.sql script. It creates a function that defines a REF cursor and returns it. Every time the testRefCur PL/SQL function is called, it opens a cursor object for the SELECT query and returns that cursor instance. To convert the object to XML, do the following:

1. Run the testRef.sql script to create the testRef package in the hr schema.
2. Compile and run the refCurTest.java program to generate XML from the results of the SQL query defined in the testRefCur function.

To apply the stylesheet, you can use the applyStylesheet command, which forces the stylesheet to be applied before generating the output.

Inserting Rows with OracleXMLSave

To insert a document into a table or view, supply the table or view name and then the document. XSU parses the document (if a string is given) and then creates an INSERT statement into which it binds all the values. By default XSU inserts values into all columns of the table or view. An absent element is treated as a NULL value. The following example shows how you can store the XML document generated from the hr.employees table in the table.

Inserting XML into All Columns with OracleXMLSave

The testInsert.java demo program inserts XML values into all columns of the hr.employees table.

The program follows these steps:

1. Create a JDBC OCI connection. The program calls the same getConnection() method used by the previous examples in this chapter:
   
   ```java
   Connection conn = getConnection("hr","password");
   ```

2. Create an XML save object. You initialize the object by passing it the Connection reference and the name of the table on which you want to perform DML. The following example illustrates this technique:

   ```java
   OracleXMLSave sav = new OracleXMLSave(conn, "employees");
   ```

3. Insert the data in an input XML document into the hr.employees table. The following code fragment creates a URL from the document filename specified on the command line:

   ```java
   sav.insertXML(sav.getURL(argv[0]));
   ```
4. Close the XML save object as follows:

   ```java
   sav.close();
   ```

**Running the testInsert Program** Assume that you write the `new_emp.xml` document to describe new employee Janet Smith, who has employee ID 7369. You pass the filename `new_emp.xml` as an argument to the testInsert program as follows:

   ```java
testInsert 'new_emp.xml'
   ```

The program inserts a new row in the `employees` table that contains the values for the columns specified. Any absent element inside the row element is treated as NULL.

**Running the program generates an INSERT statement of the following form:**

```sql
INSERT INTO hr.employees
    (employee_id, first_name, last_name, email, phone_number, hire_date,
     salary, commission_pct, manager_id, department_id)
VALUES
```

XSU matches the element tags in the input XML document that match the column names and binds their values.

**Inserting XML into a Subset of Columns with OracleXMLSave**

In some circumstances you may not want to insert values into all columns. For example, the group of values that you obtain may not be the complete set, requiring you to use triggers or default values for the remaining columns. The `testInsertSubset.java` demo program shows how to handle this case.

The program follows these steps:

1. Create a JDBC OCI connection. The program calls the same `getConnection()` method used by the previous examples in this chapter:

   ```java
   Connection conn = getConnection('hr', 'password');
   ```

2. Create an XML save object. Initialize the object by passing it the `Connection` reference and the name of the table on which you want to perform DML. The following example illustrates this technique:

   ```java
   OracleXMLSave sav = new OracleXMLSave(conn, 'employees');
   ```

3. Create an array of strings. Each element of the array should contain the name of a column in which values will be inserted. The following code fragment specifies the names of five columns:

   ```java
   String [] colNames = new String[5];
   colNames[0] = 'EMPLOYEE_ID';
   colNames[1] = 'LAST_NAME';
   colNames[2] = 'EMAIL';
   colNames[3] = 'JOB_ID';
   colNames[4] = 'HIRE_DATE';
   ```

4. Configure the XML save object to update the specified columns. The following statement passes a reference to the array to the `OracleXMLSave.setUpdateColumnList()` method:

   ```java
   sav.setUpdateColumnList(colNames);
   ```
5. Insert the data in an input XML document into the hr.employees table. The following code fragment creates a URL from the document filename specified on the command line:

```java
sav.insertXML(sav.getURL(argv[0]));
```

6. Close the XML save object as follows:

```java
sav.close();
```

**Running the testInsertSubset Program** Assume that you use the new_emp2.xml document to store data for new employee Adams, who has employee ID 7400. You pass new_emp2.xml as an argument to the testInsert program as follows:

```java
java testInsert new_emp2.xml
```

The program ignores values for the columns that were not specified in the input file. It performs an INSERT for each ROW element in the input and batches the INSERT statements by default.

The program generates the following INSERT statement:

```sql
INSERT INTO hr.employees (employee_id, last_name, email, job_id, hire_date)
VALUES (?, ?, ?, ?, ?);
```

**Updating Rows with OracleXMLSave**

To update the fields in a table or view, supply the table or view name and then the XML document. XSU parses the document (if a string is given) and then creates one or more UPDATE statements into which it binds all the values. The following examples show how you can use an XML document to update the hr.employees table.

**Updating with Key Columns with OracleXMLSave**

The testUpdate.java demo program updates the hr.employees table by invoking the OracleXMLSave.setKeyColumnList() method.

The testUpdate.java program follows these steps:

1. Create a JDBC OCI connection. The program calls the same getConnection() method used by the previous examples in this chapter:

```java
Connection conn = getConnection("hr","password");
```

2. Create an XML save object. You initialize the object by passing it the Connection reference and the name of the table on which you want to perform DML. The following example illustrates this technique:

```java
OracleXMLSave sav = new OracleXMLSave(conn, "employees");
```

3. Create a single-element String array to hold the name of the primary key column in the table to be updated. The following code fragment specifies the name of the employee_id column:

```java
String [] keyColNames = new String[1];
colNames[0] = "EMPLOYEE_ID";
```

4. Set the XML save object to the primary key specified in the array. The following statement passes a reference to the keyColNames array to the OracleXMLSave.setKeyColumnList() method:

```java
sav.setKeyColumnList(keyColNames);
```
5. Update the rows specified in the input XML document. The following statement creates a URL from the filename specified on the command line:

```java
sav.updateXML(sav.getURL(argv[0]));
```

6. Close the XML save object as follows:

```java
sav.close();
```

**Running the testUpdate Program** You can use XSU to update specified fields in a table. Example 11–2 shows upd_emp.xml, which contains updated salary and other information for the two employees that you just added, 7369 and 7400.

**Example 11–2  upd_emp.xml**

```xml
<?xml version='1.0'?>
<ROWSET>
  <ROW num='1'>
    <EMPLOYEE_ID>7400</EMPLOYEE_ID>
    <SALARY>3250</SALARY>
  </ROW>
  <ROW num='2'>
    <EMPLOYEE_ID>7369</EMPLOYEE_ID>
    <JOB_ID>SA_REP</JOB_ID>
    <MANAGER_ID>145</MANAGER_ID>
  </ROW>
  <!-- additional rows ... -->
</ROWSET>
```

For updates, supply XSU with the list of key column names in the WHERE clause of the UPDATE statement. In the hr.employees table the employee_id column is the key.

Pass the filename upd_emp.xml as an argument to the preceding program as follows:

```java
java testUpdate upd_emp.xml
```

The program generates two UPDATE statements. For the first ROW element, the program generates an UPDATE statement to update the SALARY field as follows:

```sql
UPDATE hr.employees SET salary = 3250 WHERE employee_id = 7400;
```

For the second ROW element the program generates the following statement:

```sql
UPDATE hr.employees SET job_id = 'SA_REP' AND MANAGER_ID = 145
  WHERE employee_id = 7369;
```

**Updating a Column List with OracleXMLSave**

You may want to update a table by using only a subset of the elements in an XML document. You can achieve this goal by specifying a list of columns. This technique speeds processing because XSU uses the same UPDATE statement with bind variables for all the ROW elements. You can also ignore other tags in the XML document.

**Note:** When you specify a list of columns to update, if an element corresponding to one of the update columns is absent, XSU treats it as NULL.

Suppose you want to update the salary and job title for each employee and ignore the other data. If you know that all the elements to be updated are the same for all ROW elements in the XML document, then you can use the
OracleXMLSave.setUpdateColumnNames() method to specify the columns. The testUpdateList.java program illustrates this technique.

The testUpdateList.java program follows these steps:

1. Create a JDBC OCI connection. The program calls the same getConnection() method used by the previous examples in this chapter:
   
   ```java
   Connection conn = getConnection("hr","password");
   ```

2. Create an XML save object. You initialize the object by passing it the Connection reference and the name of the table on which you want to perform DML. The following example illustrates this technique:
   
   ```java
   OracleXMLSave sav = new OracleXMLSave(conn, 'employees');
   ```

3. Create an array of type String to hold the name of the primary key column in the table to be updated. The array contains only one element, which is the name of the primary key column in the table to be updated. The following code fragment specifies the name of the employee_id column:
   
   ```java
   String [] colNames = new String[1];
   colNames[0] = "EMPLOYEE_ID";
   ```

4. Set the XML save object to the primary key specified in the array. The following statement passes a reference to the colNames array to the OracleXMLSave.setKeyColumnList() method:
   
   ```java
   sav.setKeyColumnList(keyColNames);
   ```

5. Create an array of type String to hold the name of the columns to be updated. The following code fragment specifies the name of the employee_id column:
   
   ```java
   String[] updateColNames = new String[2];
   updateColNames[0] = "SALARY";
   updateColNames[1] = "JOB_ID";
   ```

6. Set the XML save object to the list of columns to be updated. The following statement performs this task:
   
   ```java
   sav.setUpdateColumnList(updateColNames);
   ```

7. Update the rows specified in the input XML document. The following code fragment creates a URL from the filename specified on the command line:
   
   ```java
   sav.updateXML(sav.getURL(argv[0]));
   ```

8. Close the XML save object as follows:
   
   ```java
   sav.close();
   ```

Running the testUpdateList Program Suppose that you use the sample XML document upd_emp2.xml to store new data for employees Steven King, who has an employee ID of 100, and William Gietz, who has an employee ID of 206. You pass upd_emp2.xml as an argument to the testUpdateList program as follows:

```java
java testUpdateList upd_emp2.xml
```

In this example, the program generates two UPDATE statements. For the first ROW element, the program generates the following statement:

```sql
UPDATE hr.employees SET salary = 8350 AND job_id = 'AC_ACCOUNT'
WHERE employee_id = 100;
```
For the second ROW element the program generates the following statement:

```
UPDATE hr.employees SET salary = 25000 AND job_id = 'AD_PRES'
    WHERE employee_id = 206;
```

**Deleting Rows with OracleXMLSave**

When deleting from XML documents, you can specify a list of key columns. XSU uses these columns in the WHERE clause of the DELETE statement. If you do not supply the key column names, then XSU creates a new DELETE statement for each ROW element of the XML document. The list of columns in the WHERE clause of the DELETE statement matches those in the ROW element.

**Deleting by Row with OracleXMLSave**

The testDeleteRow.java demo program accepts an XML document filename as input and deletes the rows corresponding to the elements in the document.

The testDeleteRow.java program follows these steps:

1. Create a JDBC OCI connection. The program calls the same getConnection() method used by the previous examples in this chapter:
   ```java
   Connection conn = getConnection("hr", "password");
   ```

2. Create an XML save object. You initialize the object by passing it the Connection reference and the name of the table on which you want to perform DML. The following example illustrates this technique:
   ```java
   OracleXMLSave sav = new OracleXMLSave(conn, "employees");
   ```

3. Delete the rows specified in the input XML document. The following code fragment creates a URL from the filename specified on the command line:
   ```java
   sav.deleteXML(sav.getURL(argv[0]));
   ```

4. Close the XML save object as follows:
   ```java
   sav.close();
   ```

**Running the testDelete Program**

Suppose that you want to delete the employees 7400 and 7369 that you added in "Inserting Rows with OracleXMLSave" on page 11-22.

To make this example work correctly, connect to the database and disable a constraint on the hr.job_history table:

```
CONNECT hr
ALTER TABLE job_history
    DISABLE CONSTRAINT JHIST_EMP_FK;
EXIT
```

Now pass upd_emp.xml to the testDeleteRow program as follows:

```
java testDeleteRow upd_emp.xml
```

The program forms the DELETE statements based on the tag names present in each ROW element in the XML document. It executes the following statements:

```
DELETE FROM hr.employees WHERE salary = 3250 AND employee_id = 7400;
DELETE FROM hr.employees WHERE job_id = 'SA_REP' AND MANAGER_ID = 145
    AND employee_id = 7369;
```
Deleting by Key with OracleXMLSave

To only use the key values as predicates on the `DELETE` statement, invoke the `OracleXMLSave.setKeyColumnList()` method. This approach limits the number of elements used to identify a row, which has the benefit of improving performance by caching the `DELETE` statement and batching transactions. The `testDeleteKey.java` program illustrates this technique.

The `testDeleteKey.java` program follows these steps:

1. Create a JDBC OCI connection. The program calls the same `getConnection()` method used by the previous examples in this chapter:
   ```java
   Connection conn = getConnection("hr","password");
   ```

2. Create an XML save object. You initialize the object by passing it the `Connection` reference and the name of the table on which you want to perform DML. The following example illustrates this technique:
   ```java
   OracleXMLSave sav = new OracleXMLSave(conn, "employees");
   ```

3. Create an array of type `String` to hold the name of the primary key column in the table. The array contains only one element. The following code fragment specifies the name of the `employee_id` column:
   ```java
   String [] colNames = new String[1];
colNames[0] = "EMPLOYEE_ID";
   ```

4. Set the XML save object to the primary key specified in the array. The following statement passes a reference to the `colNames` array to the `OracleXMLSave.setKeyColumnList()` method:
   ```java
   sav.setKeyColumnList(keyColNames);
   ```

5. Delete the rows specified in the input XML document. The following code fragment creates a URL from the filename specified on the command line:
   ```java
   sav.deleteXML(sav.getURL(argv[0]));
   ```

6. Close the XML save object as follows:
   ```java
   sav.close();
   ```

Running the `testDeleteKey` Program  Suppose that you want to delete employees 7400 and 7369 that you added in "Updating with Key Columns with OracleXMLSave" on page 11-24. Note that if you already deleted these employees in the previous example, you can first add them back to the `employees` table as follows:

```
java testInsert new_emp.xml
java testInsert new_emp2.xml
```

Delete employees 7400 and 7369 by passing the same `upd_emp.xml` document to the `testDeleteRow` program as follows:

```
java testDeleteKey upd_emp.xml
```

The program forms the following single generated `DELETE` statement:

```
DELETE FROM hr.employees WHERE employee_id=?;
```

The program executes the following `DELETE` statements, one for each employee:

```
DELETE FROM hr.employees WHERE employee_id = 7400;
DELETE FROM hr.employees WHERE employee_id = 7369;
```
Handling XSU Java Exceptions

XSU catches all exceptions that occur during processing and throws oracle.xml.sql.OracleXMLSQLException, which is a generic runtime exception. The calling program does not have to catch this exception if it can still perform the appropriate action. The exception class provides methods to obtain error messages and also get any existing parent exception.

Obtaining the Parent Exception

The testException.java demo program throws a runtime exception and then obtains the parent exception by invoking Exception.getParentException().

Running the preceding program generates the following error message:

Caught SQL Exception:ORA-00904: "SD": invalid identifier

Raising a No Rows Exception

When there are no rows to process, XSU returns a null string. You can throw an exception every time there are no more rows, however, so that the program can process this exception through exception handlers. When a program invokes OracleXMLQuery.setRaiseNoRowsException(), XSU raises an oracle.xml.sql.OracleXMLSQLNoRowsException whenever there are no rows to generate for the output. This is a runtime exception and need not be caught.

The noRowsTest.java demo program instantiates the pageTest class defined in pageTest.java. The condition to check the termination changed from checking whether the result is null to an exception handler.

The noRowsTest.java program creates a pageTest object and initializes it with a SQL query. The program proceeds as follows:

1. Configure the query object or raise a no rows exception. The following code fragment illustrates this technique:

   pageTest test = new pageTest("SELECT * from employees");
   ...
   test.qry.setRaiseNoRowsException(true);

2. Loop through the result set infinitely, retrieving ten rows at a time. When no rows are available, the program throws an exception. The following code fragment calls pageTest.nextPage(), which scrolls through the result set ten rows at a time:

   try
   {
       while(true)
       {
           System.out.println(test.nextPage());
       }
   }

3. Catch the no rows exception and print "END OF OUTPUT". The following code illustrates this technique:

   catch(oracle.xml.sql.OracleXMLSQLNoRowsException e)
   {
       System.out.println(" END OF OUTPUT ");
       try
       {
           test.close();
       }
       catch ( Exception ae )
       {
   
   }
ae.printStackTrace(System.out);
}

After compiling the program, run it from the command line as follows:
java noRowsTest

Programming with the XSU PL/SQL API

This chapter contains the following topics:

- Generating XML from Simple Queries with DBMS_XMLQuery
- Specifying Element Names with DBMS_XMLQuery
- Paginating Results with DBMS_XMLQuery
- Setting Stylesheets in XSU
- Binding Values in XSU
- Inserting XML with DBMS_XMLSave
- Updating with DBMS_XMLSave
- Deleting with DBMS_XMLSave
- Handling Exceptions in the XSU PL/SQL API
- Reusing the Context Handle with DBMS_XMLSave

Note: For increased performance, consider using DBMS_XMLGen and DBMS_XMLStore as alternatives to DBMS_XMLQuery and DBMS_XMLSave. The two former packages are written in C and are built into the database kernel. You can also use SQL/XML functions such as XML_Element for XML access in the database.

Generating XML from Simple Queries with DBMS_XMLQuery

This section shows how you can use the DBMS_XMLQuery package to generate XML from a SQL query. To make the example work, connect to the database as hr and run the printClobOut.sql script. The script creates printClobOut, which is a simple procedure that prints a CLOB to the output buffer. If you run the printClobOut procedure in SQL*Plus, it prints the input CLOB to the screen. Set server output to ON to see the results. You may have to increase your display buffer to see all the output.

Run the simpleQuery.sql script to select 20 rows from the hr.employees table and obtain an XML document as a CLOB. The program first gets the context handle by passing in a query and then calls the getXML function to obtain the CLOB value. The document is in the same encoding as the database character set. This sample application assumes that you created the printClobOut procedure by running printClobOut.sql.

Specifying Element Names with DBMS_XMLQuery

With the XSU PL/SQL API you can change the default ROW and the ROWSET element names, which are the default names placed around each row of the result and around the whole output XML document. Use the PL/SQL procedures setRowTagName and setRowSetTagName to accomplish this task.
Connect as hr and run the changeElementName.sql script in SQL*Plus to obtain the first 20 rows of the employees table as an XML document. The anonymous PL/SQL block changes the ROW and ROWSET element names to EMP and EMPSET. Note that the block calls the printClobOut procedure that you created by running printClobOut.sql.

The generated XML document has an <EMPSET> document element. Each row is separated with the <EMP> tag.

Paginating Results with DBMS_XMLQuery

You can paginate query results by calling the following PL/SQL functions:

- setMaxRows sets the maximum number of rows to be converted to XML. This maximum is relative to the current row position from which the previous result was generated.
- setSkipRows specifies the number of rows to skip before converting the row values to XML.

Run the paginateResult.sql script to execute an anonymous block that paginates results. It skips the first 3 rows of the employees table and prints the rest of the rows 10 at a time by setting skipRows to 3 for the first batch of 10 rows and then to 0 for the rest of the batches. For multiple fetches, you must determine when there are no more rows to fetch, which you can do by calling setRaiseNoRowsException. This procedure raises an exception if no rows are written to the CLOB. This exception can be caught and used as the termination condition.

Setting Stylesheets in XSU

The XSU PL/SQL API provides the ability to set stylesheets on the generated XML documents as follows:

- Set the stylesheet header in the result with the setStyleSheetHeader procedure. This procedure adds the XML processing instruction that includes the stylesheet.
- Apply a stylesheet to the resulting XML document before generation. This method increases performance dramatically because otherwise the XML document must be generated as a CLOB, sent to the parser again, and have the stylesheet applied. XSU generates a DOM document, calls the parser, applies the stylesheet and then generates the result. To apply the stylesheet to the resulting XML document, use the setXSLT procedure, which uses the stylesheet to generate the result.

Binding Values in XSU

The XSU PL/SQL API provides the ability to bind values to a SQL statement. The SQL statement can contain named bind variables, which must be prefixed with a colon (:).

The bindSQLVariables.sql script runs an anonymous PL/SQL block that binds values for EMPLOYEE_ID and FIRST_NAME to columns in the employees table.

Inserting XML with DBMS_XMLSave

To insert a document into a table or view, supply the table or the view name and then the XML document. XSU parses the XML document (if a string is given) and then creates an INSERT statement into which it binds all the values. By default, XSU inserts values into all the columns of the table or view and treats absent elements as NULL.
Inserting Values into All Columns with DBMS_XMLSave

Run the insProc.sql demo script to create a PL/SQL stored procedure, insProc, which accepts the following parameters:

- An XML document as a CLOB
- The name of the table in which to insert the document

You can invoke the insProc procedure to insert an XML document into the table.

Run the insertClob.sql script to create a table called xmldocument and store an XML document in the table as a CLOB. The XML document describes employee 7370, Liz Gardner, whom you want to insert into the hr.employees table.

Example 11–3  insertClob.sql

```sql
CREATE TABLE hr.xmldocument
    (docid NUMBER PRIMARY KEY,
     xml_text CLOB);
-- insert an XML document into the CLOB column
INSERT INTO hr.xmldocument (docid,xml_text) VALUES (1,
    '<?xml version="1.0"?>
    <ROWSET>
    <ROW num="1"><EMPLOYEE_ID>7370</EMPLOYEE_ID><FIRST_NAME>Liz</FIRST_NAME><LAST_NAME>Gardner</LAST_NAME><EMAIL>liz.gardner@business.com</EMAIL><PHONE_NUMBER>650-555-6127</PHONE_NUMBER><HIRE_DATE>12/18/2004 0:0:0</HIRE_DATE><SALARY>3000</SALARY><COMMISSION_PCT>0</COMMISSION_PCT><JOB_ID>SH_CLERK</JOB_ID><MANAGER_ID>103</MANAGER_ID><DEPARTMENT_ID>20</DEPARTMENT_ID></ROW></ROWSET>');```

Run the insertEmployee.sql script shown in Example 11–4 to call the insProc stored procedure and insert Liz Gardner into the employees table.

Example 11–4  insertEmployee.sql

```sql
DECLARE
    v_xml_text CLOB;
BEGIN
    SELECT xml_text INTO v_xml_text FROM hr.xmldocument WHERE docid = 1;
    insProc(v_xml_text, 'employees');
END;
/
```

As in "Inserting Rows with OracleXMLSave" on page 11-22, running the callinsProc procedure generates an INSERT statement of the form shown in Example 11–5.

Example 11–5  Form of the INSERT Statement

```sql
INSERT INTO hr.employees
```
(employee_id, first_name, last_name, email, phone_number, hire_date, salary, commission_pct, manager_id, department_id)
VALUES

XSU matches the element tags in the input XML document that match the column names and binds their values.

**Inserting into a Subset of Columns with DBMS_XMLSave**

As explained in "Inserting XML into a Subset of Columns with OracleXMLSave" on page 11-23, you may not want to insert values into all columns. You can create a list of column names for insert processing and pass it to the `DBMS_XMLSave` procedure. You can set these values by calling the `setUpdateColumnName` procedure repeatedly and passing in a column name to update every time. Clear the column name settings by invoking `clearUpdateColumnList`.

Run the `testInsert.sql` demo script to create a PL/SQL stored procedure called `testInsert`. You can use this procedure to insert XML data of type CLOB into the `hr.employees` table.

Run the `insertClob2.sql` script shown in Example 11–6 to insert an XML document describing new employee Jordan into a CLOB column of the `xmldocument` table. Note that the document does not contain an element corresponding to every column in the `employees` table.

**Example 11–6  insertClob2.sql**

```sql
-- insert an XML document into the CLOB column of the xmldocument table with only
-- some of the possible elements
INSERT INTO hr.xmldocument (docid, xml_text)
VALUES (2,
  '<?xml version='1.0'?>
  <ROWSET>
  <ROW num='1'>
    <EMPLOYEE_ID>7401</EMPLOYEE_ID>
    <LAST_NAME>Jordan</LAST_NAME>
    <EMAIL>jim.jordan@business.com</EMAIL>
    <JOB_ID>SH_CLERK</JOB_ID>
    <HIRE_DATE>12/17/2004 0:0:0</HIRE_DATE>
  </ROW>
  </ROWSET>');
```

Running the `insertEmployee2.sql` script shown in Example 11–7 inserts the data for employee Jim Jordan into a subset of the columns in the `hr.employees` table.

**Example 11–7  insertEmployee2.sql**

```sql
DECLARE
  v_xml_text CLOB;
BEGIN
  SELECT xml_text
  INTO v_xml_text
  FROM hr.xmldocument
  WHERE docid = 2;
  testInsert(v_xml_text);
END;
/
```
As in "Inserting XML into a Subset of Columns with OracleXMLSave" on page 11-23, calling testInsert generates the following INSERT statement:

```
INSERT INTO hr.employees (employee_id, last_name, email, job_id, hire_date)
VALUES (?, ?, ?, ?, ?);
```

** Updating with DBMS_XMLSave **

As described in "Updating Rows with OracleXMLSave" on page 11-24, you can use an XML document to update specified fields in a table. You can either specify a column to use as a key or pass a list of columns for updating.

** Updating with Key Columns with DBMS_XMLSave **

Run the testUpdateKey.sql script to create a PL/SQL procedure called testUpdateKey. This procedure uses the employee_id column of the hr.employees table as a primary key.

Run the insertClob3.sql script shown in shown in Example 11–8 to insert an XML document into the CLOB column of the xmldocument table. The document specifies a new salary for employee 7400 and a new job ID and manager ID for employee 7369.

** Example 11–8 insertClob3.sql **

```
INSERT INTO hr.xmldocument (docid, xml_text)
VALUES (3,
       '<?xml version="1.0"?>
       <ROWSET>
           <ROW num="1">
               <EMPLOYEE_ID>7400</EMPLOYEE_ID>
               <SALARY>3250</SALARY>
           </ROW>
           <ROW num="2">
               <EMPLOYEE_ID>7369</EMPLOYEE_ID>
               <JOB_ID>SA_REP</JOB_ID>
               <MANAGER_ID>145</MANAGER_ID>
            </ROW>
        </ROWSET>);
```

Run the updateEmployee.sql script shown in Example 11–9 to pass the XML document to the testUpdateKey procedure and generate two UPDATE statements.

** Example 11–9 updateEmployee.sql **

```
DECLARE
    v_xml_text CLOB;
BEGIN
    SELECT xml_text
          INTO v_xml_text
          FROM hr.xmldocument
          WHERE docid = 3;
    testUpdateKey(v_xml_text);
END;
/
```

For the first ROW element, the program generates an UPDATE statement as follows:

```
UPDATE hr.employees SET salary = 3250 WHERE employee_id = 7400;
```

For the second ROW element the program generates the following statement:
UPDATE hr.employees SET job_id = 'SA_REP' AND MANAGER_ID = 145
WHERE employee_id = 7369;

Specifying a List of Columns with DBMS_XMLSave: Example
As described in “Updating a Column List with OracleXMLSave” on page 11-25, you can specify a list of columns to update.

Run the testUpdateSubset.sql script creates the PL/SQL procedure testUpdateSubset. The procedure uses the employee_id column as key and updates only the salary and job_id columns of the hr.employees table.

Run the insertClob4.sql script to insert an XML document into the xmldocument table. The <ROW> elements in the document describe employees 100 and 206. Each <ROW> element has ten subelements that contain descriptive text.

Run the updateEmployee2.sql script shown in Example 11–10 to pass the XML CLOB to the testUpdateSubset procedure and generate two UPDATE statements.

Example 11–10  updateEmployee2.sql

DECLARE
  v_xml_text CLOB;
BEGIN
  SELECT xml_text
  INTO v_xml_text
  FROM hr.xmldocument
  WHERE docid = 4;
  testUpdateSubset(v_xml_text);
END;
/

The procedure updates only those columns specified in the setUpdateColumn procedure, salary and email, for employees 100 and 206.

Deleting with DBMS_XMLSave
As described in “Deleting Rows with OracleXMLSave” on page 11-27, you can supply a list of key columns that XSU uses to determine which rows to delete. XSU specifies these columns in the WHERE clause of the DELETE statement.

Deleting by Row with DBMS_XMLSave: Example
Create the testDeleteRow PL/SQL procedure by running the testDeleteRow.sql script. The procedure deletes a row from the hr.employees table for every <ROW> element in an input XML document.

Suppose that you want to delete the employee Jim Jordan that you added in Example 11–7 on page 11-33. Run the deleteEmployeeByRow.sql script shown in Example 11–11 to pass the XML document as a CLOB to the testDeleteRow stored procedure.

Example 11–11  Deleting by Row

DECLARE
  v_xml_text CLOB;
BEGIN
  SELECT xml_text
  INTO v_xml_text
  FROM hr.xmldocument
  WHERE docid = 2;
testDeleteRow(v_xml_text);
END;
/

The preceding call to testDeleteRow generates the following DELETE statement:

```
DELETE FROM hr.employees
    WHERE employee_id = 7401 AND last_name = 'JORDAN'
    AND email = 'jim.jordan@business.com' AND job_id = 'SH_CLERK'
    AND hire_date = '12/17/2004 0:0:0';
```

The program forms the DELETE statements based on the tag names present in each `<ROW>` element in the XML document.

**Deleting by Key with DBMS_XMLSave: Example**

As explained in "Deleting by Key with OracleXMLSave" on page 11-28, you can specify a column to use as a primary key for the deletions. Use the DBMS_XMLSave.setKeyColumn function to specify the key.

The testDeleteKey procedure created by running testDeleteKey.sql deletes a row from the employees table for every `<ROW>` element in an input XML document.

Suppose that you want to delete the employee Liz Gardner that you added in Example 11–4 on page 11-32. Run the deleteEmployeeByKey.sql script shown in Example 11–12 to pass the XML document as a CLOB to the testDeleteKey stored procedure.

**Example 11–12 Deleting by Key**

```sql
DECLARE
    v_xml_text CLOB;
BEGIN
    SELECT xml_text
    INTO v_xml_text
    FROM hr.xmldocument
    WHERE docid = 1;
    testDeleteKey(v_xml_text);
END;
/
```

In the preceding procedure call, XSU generates a single DELETE statement of the following form:

```
DELETE FROM hr.employees WHERE employee_id=?
```

XSU uses this statement for all `<ROW>` elements in the input XML document.

**Handling Exceptions in the XSU PL/SQL API**

Good PL/SQL coding practice accounts for possible exceptions. The anonymous PL/SQL block in raiseException.sql demonstrates how to invoke the DBMS_XMLQuery.getExceptionContent procedure. Run the script in SQL*Plus to print the following error message:

```
Exception caught 904 ORA-00904: "Z": invalid identifier
```

**Reusing the Context Handle with DBMS_XMLSave**

In the DML examples described in the preceding sections, you can use the same context handle to perform more than one operation. That is, you can perform more
than one INSERT with the same context provided that all of the insertions access the same table specified when creating the save context. You can also use the same context to mix DML statements.

The testDML.sql script shows how to use the same context and settings to perform DML depending on user input. The example uses a PL/SQL supplied package static variable to store the context so that the same context can be used for all function calls.

In the testDML package created by the script, you create a context once for the whole package (and thus the session) and reuse the context for multiple DML operations.

**Note:** The key column employee_id is used both for updates and deletes as a way of identifying the row.

You can call any of the three procedures created by the script to update the employees table:

```java
testDML.insertXML(xmlclob);
testDML.deleteXML(xmlclob);
testDML.updateXML(xmlclob);
```

Each procedure call uses the same context, which improves the performance of these operations, particularly if these operations are performed frequently.

**Tips and Techniques for Programming with XSU**

This section provides additional tips and techniques for writing programs with XSU. It contains the following topics:

- How XSU Maps Between SQL and XML
- How XSU Processes SQL Statements

**How XSU Maps Between SQL and XML**

The fundamental component of a table is a column, whereas the fundamental components of an XML document are elements and attributes. How do tables map to XML documents? For example, if the hr.employees table has a column called last_name, how is this structure represented in XML: as an <EMPLOYEES> element with a last_name attribute or as a <LAST_NAME> element within a different root element? This section answers such questions by describing how SQL maps to XML and vice versa. It contains the following topics:

- Default SQL to XML Mapping
- Default XML to SQL Mapping
- Customizing Generated XML

**Default SQL to XML Mapping**

Assume that you want to display data from some column of the hr.employees table as an XML document. You run XSU at the command line as follows:

```bash
java OracleXML getXML -user "hr/password" -withschema \
"SELECT employee_id, last_name, hire_date FROM employees"
```
XSU outputs an XML document based on the input query. The root element of the
document is <DOCUMENT>. The following shows sample output, with extraneous lines
replaced by comments:

```xml
<?xml version = '1.0'?>
<DOCUMENT xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <!-- children of schema element ... -->
    </xsd:schema>
    <ROWSET xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="#/DOCUMENT/xsd:schema[not(@targetNamespace)]">
        <ROW num="1">
            <EMPLOYEE_ID>100</EMPLOYEE_ID>
            <LAST_NAME>King</LAST_NAME>
            <HIRE_DATE>6/17/1987 0:0:0</HIRE_DATE>
        </ROW>
        <!-- additional rows ... -->
    </ROWSET>
</DOCUMENT>
```

In the generated XML, the rows returned by the SQL query are children of the
<ROWSET> element. The XML document has the following features:

- The <ROWSET> element has zero or more <ROW> child elements corresponding to the
  number of rows returned. If the query generates no rows, then no <ROW> elements
  are included; if the query generates one row, then one <ROW> element is included,
  and so forth.

- Each <ROW> element contains data from one of the table rows. Specifically, each
  <ROW> element has one or more child elements whose names and content are
  identical to the database columns specified in the SELECT statement.

**XML Mapping Against an Object-Relational Schema**
Assume a case in which you generate an XML document from an object-relational schema. Run the `createObjRelSchema.sql`
script in SQL*Plus to set up and populate an object-relational schema. The schema
contains a `dept1` table with two columns that employ user-defined types.

You can query the `dept1` table as follows by invoking XSU from the command line:

```
% java OracleXML getXML -user "hr/password" -withschema "SELECT * FROM dept1"
```

XSU returns the XML document shown in **Example 11–13**, which is altered so that
extraneous lines are replaced by comments.

**Example 11–13   XSU-Generated Sample Document**

```xml
<?xml version='1.0'?><
<DOCUMENT xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:schema targetNamespace="http://xmlns.oracle.com/xdb/SYSTEM"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:SYSTEM="http://xmlns.oracle.com/xdb/SYSTEM">
        <!-- children of schema element ... -->
    </xsd:schema>
    <ROWSET xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="#/DOCUMENT/xsd:schema[not(@targetNamespace)]">
        <ROW num="1">
            <DEPTNO>120</DEPTNO>
            <DEPTNAME>Treasury</DEPTNAME>
            <DEPTADDR>
                <STREET>2004 Charade Rd</STREET>
                <CITY>Seattle</CITY>
        </ROW>
```

```xml
</ROWSET>
</DOCUMENT>
```
<STATE>WA</STATE>
<ZIP>98199</ZIP>
</DEPTADDR>
<EMPLIST>
<EMPLIST_ITEM>
<EMPLOYEE_ID>1</EMPLOYEE_ID>
-LAST_NAME>Mehta</LAST_NAME>
<SALARY>6000</SALARY>
<EMPLOYEE_ADDRESS>
-STREET>500 Main Road</STREET>
-CITY>Seattle</CITY>
<STATE>WA</STATE>
<ZIP>98199</ZIP>
</EMPLOYEE_ADDRESS>
</EMPLIST_ITEM>
</EMPLIST>
</ROW>
</ROWSET>
</DOCUMENT>

As in the previous example, the mapping is canonical, that is, <ROWSET> contains <ROW> child elements, which in turn contain child elements corresponding to the columns in dept1. For example, the <DEPTNAME> element corresponds to the dept1.deptname column. The elements corresponding to scalar type columns contain the data from the columns.

Default Mapping of Complex Type Columns to XML The situation is more complex with elements corresponding to a complex type column. In Example 11–13, <DEPTADDR> corresponds to the dept1.deptAddr column, which is of object type AddressType. Consequently, <DEPTADDR> contains child elements corresponding to the attributes specified in the type AddressType. The AddressType attribute street corresponds to the child XML element <STREET> and so forth. These sub-elements can contain data or subelements of their own, depending on whether the attribute they correspond to is of a simple or complex type.

Default Mapping of Collections to XML When dealing with elements corresponding to database collections, the situation is also different. In Example 11–13, the <EMPLIST> element corresponds to the emplist column of type EmployeeListType. Consequently, the <EMPLIST> element contains a list of <EMPLIST_ITEM> elements, each corresponding to one of the elements of the collection. Note the following:

- The <ROW> elements contain a cardinality attribute num.
- If a particular column or attribute value is NULL, then for that row, the corresponding XML element is left out altogether.
- If a top-level scalar column name starts with the at sign (@) character, then the column is mapped to an XML attribute instead of an XML element.

Default XML to SQL Mapping XML to SQL mapping is the reverse of SQL to XML mapping. Consider the following differences when using XSU to map XML to SQL:

- When transforming XML to SQL, XSU ignores XML attributes. Thus, there is really no mapping of XML attributes to SQL.
- When transforming SQL to XML, XSU performs the mapping on a single ResultSet created by a SQL query. The query can span multiple database tables or views. When transforming XML into SQL, note the following:
To insert one XML document into multiple tables or views, you must create an object-relational view over the target schema.

If the view is not updatable, then you can use **INSTEAD OF INSERT** triggers.

If the XML document does not perfectly map to the target database schema, then you can perform the following actions:

- Modify the target. Create an object-relational view over the target schema and make the view the new target.
- Modify the XML document by using XSLT to transform the XML document. You can register the XSLT stylesheet with XSU so that the incoming XML is automatically transformed before it attempts any mapping.
- Modify XSU’s XML-to-SQL mapping. You can instruct XSU to perform case-insensitive matching of XML elements to database columns or attributes. For example, you can instruct XSU to do the following:
  - Use the name of the element corresponding to a database row instead of **ROW**.
  - Specify the date format to use when parsing dates in the XML document.

### Customizing Generated XML

In some circumstances you may need to generate XML with a specific structure. Because the desired structure may differ from the default structure of the generated XML document, you want to have some flexibility in this process. You can customize the structure of a generated XML document by using one of the following methods:

- **Altering the Database Schema or SQL Query**
- **Modifying XSU**

### Altering the Database Schema or SQL Query

You can perform source customizations by altering the SQL query or the database schema. The simplest and most powerful source customizations include the following:

- In the database schema, create an object-relational view that maps to the desired XML document structure.
- In your query, do the following:
  - Use cursor subqueries or cast-multiset constructs to create nesting in the XML document that comes from a flat schema.
  - Alias column and attribute names to obtain the desired XML element names.
  - Alias top-level scalar type columns with identifiers that begin with the at sign (@) to make them map to an XML attribute instead of an XML element. For example, executing the following statement generates an XML document in which the `<ROW>` element has the attribute `empno`:
    ```sql
    SELECT employee_name AS "@empno", ... FROM employees;
    ```

Consider the `customer.xml` document shown in Example 11–14.

**Example 11–14  customer.xml**  
```xml
<?xml version="1.0"?>
<ROWSET>
  <ROW num="1">
    <CUSTOMER>
      <CUSTOMERID>1044</CUSTOMERID>
      <NAME>XYZ</NAME>
      <ADDRESS>123 Main St</ADDRESS>
    </CUSTOMER>
  </ROW>
</ROWSET>
```
Suppose that you need to design a set of database tables to store this data. Because the XML is nested more than one level, you can use an object-relational database schema that maps canonically to the preceding XML document. Run the createObjRelSchema2.sql script in SQL*Plus to create such a database schema.

You can load the data in the customer.xml document into the customer_tab table created by the script. Invoke XSU for Java from the command line as follows:

```
java OracleXML putXML -user "hr/password" -fileName customer.xml customer_tab
```

To load customer.xml into a database schema that is not object-relational, you can create objects in views on top of a standard relational schema. For example, you can create a relational table that contains the necessary columns, then create a customer view that contains a customer object on top of it, as shown in the createRelSchema.sql script in Example 11–15.

### Example 11–15  createRelSchema.sql

```sql
CREATE TABLE hr.cust_tab

( customerid NUMBER(10),
  firstname VARCHAR2(20),
  lastname VARCHAR2(20),
  street VARCHAR2(40),
  city VARCHAR2(20),
  state VARCHAR2(20),
  zip VARCHAR2(20)
);

CREATE VIEW customer_view

AS

SELECT customer_type(customerid, firstname, lastname,
  address_type(street,city,state,zip))

  customer

FROM cust_tab;
```

You can load data into customer_view as follows:

```
java OracleXML putXML -user "hr/password" -fileName customer.xml customer_view
```

Alternatively, you can flatten your XML by means of XSLT and then insert it directly into a relational schema. However, this is the least recommended option.

Suppose that you want to map a particular column or a group of columns to an XML attribute instead of an XML element. To achieve this functionality, you can create an alias for the column name and prepend the at sign (@) before the name of this alias. For example, you can use the mapColumnToAtt.sql script to query the hr.employees table, rendering employee_id as an XML attribute.

You can run the mapColumnToAtt.sql script from the command line as follows:
java OracleXML getXML -user "hr/password" -fileName "mapColumnToAtt.sql"

**Note:** All attributes must appear before any non-attribute.

**Modifying XSU** XSU enables you to modify the rules that it uses to transform SQL data into XML. You can make any of the following changes when mapping SQL to XML:

- Change or omit the `<ROWSET>` or `<ROW>` tag.
- Change or omit the attribute `num`, which is the cardinality attribute of the `<ROW>` element.
- Specify the case for the generated XML element names.
- Specify that XML elements corresponding to elements of a collection must have a cardinality attribute.
- Specify the format for dates in the XML document.
- Specify that null values in the XML document have to be indicated with a nullness attribute rather than by omission of the element.

**How XSU Processes SQL Statements**

This section describes how XSU interacts with the database:

- How XSU Queries the Database
- How XSU Inserts Rows
- How XSU Updates Rows
- How XSU Deletes Rows
- How XSU Commits After DML

**How XSU Queries the Database**

XSU executes SQL queries and retrieves the `ResultSet` from the database. XSU then acquires and analyzes metadata about the `ResultSet`. Using the mapping described in "Default SQL to XML Mapping" on page 11-37, XSU processes the SQL result set and converts it into an XML document.

XSU cannot handle certain types of queries, especially those that mix columns of type `LONG` or `LONG RAW` with `CURSOR()` expressions in the `SELECT` clause. `LONG` and `LONG RAW` are two examples of datatypes that JDBC accesses as streams and whose use is deprecated. If you migrate these columns to `CLOBs`, then the queries succeed.

**How XSU Inserts Rows**

When inserting the contents of an XML document into a table or view, XSU performs the following steps:

1. Retrieves metadata about the target table or view.
2. Generates a SQL `INSERT` statement based on the metadata. For example, assume that the target table is `dept1` and the XML document is generated from `dept1`. XSU generates the following `INSERT` statement:

   ```java
   INSERT INTO dept1 (deptno, deptname, deptaddr, emplist) VALUES (?, ?, ?, ?)
   ```
3. Parses the XML document, and for each record, it binds the appropriate values to the appropriate columns or attributes. For example, it binds the values for **INSERT** statement as follows:

```
depnto <= 100
depntname <= 'SPORTS'
depntaddr <= AddressType('100 Redwood Shores Pkwy', 'Redwood Shores', 'CA', '94065')
emlist <= EmployeeListType(EmployeeType(7369, 'John', 100000, AddressType('300 Embarcadero', 'Palo Alto', 'CA', '94056'),...)
```

4. Executes the statement. You can optimize **INSERT** processing to insert in batches and commit in batches.

**See Also:**
- "Default SQL to XML Mapping" on page 11-37
- "Inserting Rows with OracleXMLSave" on page 11-22 for more detail on batching

**How XSU Updates Rows**

Updates and delete statements differ from inserts in that they can affect more than one row in the database table. For inserts, each `<ROW>` element of the XML document can affect at most one row in the table if no triggers or constraints are on the table. With updates and deletes, the XML element can match more than one row if the matching columns are not key columns in the table.

For update statements, you must provide a list of key columns that XSU needs to identify the row to update. For example, assume that you have an XML document that contains the following fragment:

```
<ROWSET>
  <ROW num='1'>
    <DEPTNO>100</DEPTNO>
    <DEPTNAME>SportsDept</DEPTNAME>
  </ROW>
</ROWSET>
```

You want to change the **DEPTNAME** value from **Sports** to **SportsDept**. If you supply the **DEPTNO** as the key column, then XSU generates the following **UPDATE** statement:

```
UPDATE dept1 SET deptname = ? WHERE deptno = ?
```

XSU binds the values in the following way:

```
depntno <= 100
depntname <= 'SportsDept'
```

For updates, you can also choose to update only a set of columns and not all the elements present in the XML document.

**See Also:** "Updating Rows with OracleXMLSave" on page 11-24

**How XSU Deletes Rows**

For deletes, you can choose to provide a set of key columns so that XSU can identify the rows to be deleted. If you do not provide the set of key columns, then the **DELETE** statement tries to match all the columns in the document. Assume that you pass the following document to XSU:
<ROWSET>
  <ROW num="1">
    <DEPTNO>100</DEPTNO>
    <DEPTNAME>Sports</DEPTNAME>
    <DEPTADDR>
      <STREET>100 Redwood Shores Pkwy</STREET>
      <CITY>Redwood Shores</CITY>
      <STATE>CA</STATE>
      <ZIP>94065</ZIP>
    </DEPTADDR>
  </ROW>
  <!-- additional rows ... -->
</ROWSET>

XSU builds a DELETE statement for each ROW element:

DELETE FROM dept1 WHERE deptno = ? AND deptname = ? AND deptaddr = ?

The binding is as follows:

deptno   <- 100  
deptname <- sports  
deptaddr <- addresstype('100 redwood shores pkwy','redwood city','ca','94065')

See Also: "Deleting Rows with OracleXMLSave" on page 11-27

How XSU Commits After DML

By default XSU performs no explicit commits. If AUTOCOMMIT is on, which is the default for a JDBC connection, then after each batch of statement executions XSU executes a COMMIT. You can override this behavior by turning AUTOCOMMIT off and then using setCommitBatch to specify the number of statement executions before XSU should commit. If an error occurs, then XSU rolls back to either the state the target table was in before the call to XSU, or the state after the last commit made during the current call to XSU.
This chapter contains these topics:

- Introduction to the TransX Utility
- Using the TransX Utility: Overview
- Loading Data with the TransX Utility

See Also: Chapter 13, "Data Loading Format (DLF) Specification"

Introduction to the TransX Utility

**TransX Utility** enables you to transfer XML to a database. More specifically, the TransX utility is an application of **XML SQL Utility (XSU)** that loads translated seed data and messages into a database schema. If you have data to be populated into a database in multiple languages, then the utility provides the functionality that you would otherwise need to develop with XSU.

The TransX utility is particularly useful when handling multilingual XML. The utility does the following:

- Automatically manages the change variables, start sequences, and additional SQL statements that would otherwise require multiple inserts or sessions. Thus, translation vendors do not need to work with unfamiliar SQL and PL/SQL scripts.
- Automates character encoding. Consequently, loading errors due to incorrect encoding are impossible so long as the data file conforms to the XML standard.
- Reduces globalization costs by preparing strings to be translated, translating the strings, and loading them into the database.
- Minimizes translation data format errors and accurately loads the translation contents into pre-determined locations in the database. When the data is in a predefined format, the TransX utility validates it.
- Eliminates syntax errors due to varying Globalization Support settings.
- Does not require the `UNISTR` construct for every piece of `NCHAR` data.

Prerequisites

This chapter assumes that you are familiar with **XML SQL Utility (XSU)** because TransX is an application of XSU.

See Also: Chapter 11, "Using the XML SQL Utility (XSU)"
TransX Utility Features

This section describes the following features of the TransX utility:

- Simplified Multilingual Data Loading
- Simplified Data Format Support and Interface
- Additional TransX Utility Features

Simplified Multilingual Data Loading

When inserting multilingual data or data translations into an Oracle database, or when encoding, each XML file requires validation. The traditional translation data loading method is to change the `NLS_LANG` environment variable setting when switching load files. This variable sets the language and territory used by the client application and the database server. It also sets the client character set, which is the character set for data entered or displayed by a client program.

In the traditional method, each load file is encoded in a character set suitable for its language, which is necessary because translations must be performed in the same file format—typically in a SQL script—as the original. The `NLS_LANG` setting changes as files are loaded to adapt to the character set that corresponds to the language. As well as consuming time, this approach is error-prone because the encoding metadata is separate from the data itself.

With the TransX utility you use an XML document with a predefined format called a dataset. The dataset contains the encoding information and the data so that you can transfer multilingual data without changing `NLS_LANG` settings. The TransX utility frees development and translation groups by maintaining the correct character set while loading XML data into the database.

See Also: Oracle Database Globalization Support Guide to learn about the `NLS_LANG` environment variable

Simplified Data Format Support and Interface

The TransX utility provides a command-line interface and programmable API. The utility complies with a data format defined to be the canonical method for the representation of seed data loaded into the database. The format is intuitive and simplified for use by translation groups. The format specification defines how translators can describe the data so that it is loaded in an expected way. You can represent the values in the data set with scalar values or expressions such as constants, sequences, and queries.

Additional TransX Utility Features

Table 12–1 describes other useful TransX utility features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>TransX Utility . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-line interface</td>
<td>Provides easy-to-use commands.</td>
</tr>
<tr>
<td>User API</td>
<td>Exposes a Java API.</td>
</tr>
<tr>
<td>Validation</td>
<td>Validates the data format and reports errors.</td>
</tr>
<tr>
<td>Whitespace handling</td>
<td>Does not consider whitespace characters in the data set as significant unless otherwise specified in various granularity.</td>
</tr>
</tbody>
</table>
Using the TransX Utility: Overview

This section contains the following topics:

- Using the TransX Utility: Basic Process
- Running the TransX Utility Demo Programs
- Using the TransX Command-Line Utility

Using the TransX Utility: Basic Process

TransX is accessible through the following API:

- `oracle.xml.transx.loader` class, which contains the `getLoader()` method to obtain a TransX instance
- `oracle.xml.transx.TransX` interface, which is the TransX API

Figure 12–1 illustrates the basic process for using the TransX API to transfer XML to an Oracle database.

![Figure 12–1. Basic Process of a TransX Application](image)

The basic process of a TransX application is as follows:

1. Create a TransX loader object. Instantiate the `TransX` class by calling `getLoader()` as follows:

   ```java
   TransX transx = loader.getLoader();
   ```
2. Start a data loading session by supplying database connection information with `TransX.open()`. You create a session by supplying the JDBC connect string, database username, and database password. You have the following options:

- Create the connection with the JDBC OCI driver. The following code fragment illustrates this technique and connects with the supplied user name and password:

  ```java
  transx.open( "jdbc:oracle:oci8:@", user, passwd );
  ```

- Create the connection with the JDBC thin driver. The thin driver is written in pure Java and can be called from any Java program. The following code fragment illustrates this technique and connects:

  ```java
  transx.open( "jdbc:oracle:thin:@//myhost:1521/myservicename", user, passwd );
  ```

  The thin driver requires the host name (`myhost`), port number (`1521`), and the service name (`myservicename`). The database must have an active TCP/IP listener.

**Note:** If you are just validating your data format, then you do not need to establish a database connection because the validation is performed by TransX. Thus, you can invoke the `TransX.validate()` method without a preceding `open()` call.

3. Configure the TransX loader. Table 12–2 describes configuration methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setLoadingMode()</code></td>
<td>Sets the operation mode on duplicates. The mode determines TransX behavior when there are one or more existing rows in the database whose values in the key columns are the same as those in the dataset to be loaded. You can specify the constants <code>EXCEPTION_ON_DUPLICATES</code>, <code>SKIP_DUPLICATES</code>, or <code>UPDATE_DUPLICATES</code> in class <code>oracle.xml.transx.LoadingMode</code>. By default the loader skips duplicates.</td>
</tr>
<tr>
<td><code>setNormalizeLangTag()</code></td>
<td>Sets the case of language tag. By default the loader uses the style specified in the <code>normalize-langtag</code> attribute on DLF.</td>
</tr>
<tr>
<td><code>setPreserveWhitespace()</code></td>
<td>Specifies how the loader should handle whitespace. The default is <code>FALSE</code>, which means that the loader ignores the type of whitespace characters in the dataset and loads them as space characters. The loader treats consecutive whitespace characters in the dataset as one space character.</td>
</tr>
<tr>
<td><code>setValidationMode()</code></td>
<td>Sets the validation mode. The default is <code>TRUE</code>, which means that the loader performs validation of the dataset format against the canonical schema definition on each <code>load()</code> call. The validation mode should be disabled only if the dataset has already been validated.</td>
</tr>
</tbody>
</table>

The following example specifies that the loader should skip duplicate rows and not validate the dataset:

```java
transx.setLoadingMode( LoadingMode.SKIP_DUPLICATES );
transx.setValidationMode( false );
```
4. Load the datasets by invoking TransX.load(). The same JDBC connection is used during the iteration of the load operations. For example, load three datasets as follows:

```java
String datasrc[] = {"data1.xml", "data2.xml", "data3.xml"};
...
for ( int i = 0 ; i < datasrc.length ; i++ )
{
    transx.load( datasrc[i] );
}
```

5. Close the loading session by invoking TransX.close(). This method call closes the database connection:

```java
transx.close();
```

See Also:
- Oracle Database Java Developer’s Guide to learn about Oracle JDBC
- Oracle Database XML Java API Reference to learn about TransX classes and methods

Running the TransX Utility Demo Programs

Demo programs for the TransX utility are included in $ORACLE_HOME/xdk/demo/java/transx. Table 12–3 describes the XML files and programs that you can use to test the utility.

### Table 12–3 TransX Utility Sample Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>README</td>
<td>A text file that describes how to set up the TransX demos.</td>
</tr>
</tbody>
</table>
| emp-dlf.xml   | A sample output file. The following command generates a file emp.xml that contains all data in the table emp: transx -s 'jdbc:oracle:thin:@//myhost:1521/myservicename" user -pw emp.xml emp  
The emp-dlf.xml file should be identical to emp.xml. |
| txclean.sql   | A SQL file that drops the tables and sequences created for the demo. |
| txdemo1.java  | A sample Java application that creates a JDBC connection and loads three datasets into the database. |
| txdemo1.sql   | A SQL script that creates two tables and a sequence for use by the demo application. |
| txdemo1.xml   | A sample dataset. |

Documentation for how to compile and run the sample programs is located in the README. The basic steps are as follows:

1. Change into the $ORACLE_HOME/xdk/demo/java/transx directory (UNIX) or %ORACLE_HOME%/xdk/demo/java/transx directory (Windows).

2. Make sure that your environment variables are set as described in “Setting Up the Java XDK Environment” on page 3-5. It is recommended that you set the $ORACLE_SID (UNIX) or %ORACLE_SID% (Windows) environment variables to the default database.
3. Set up the sample database objects by executing txdemo1.sql. Connect to the database and run the txdemo1.sql script as follows:

```
txdemo1
```

4. Run the TransX utility from the command line. For example, assume that you want to connect with the Java thin driver and that your host is myhost, your port is 1521, and your service name is myservicename. Enter the user name where the token user appears. You can execute the following command to load dataset txdemo1.xml:

```
transx "jdbc:oracle:thin:@//myhost:1521/myservicename" user -pw txdemo1.xml
```

When the operation is successful, nothing is printed out on your terminal.

5. Query the database to determine whether the load was successful. For example:

```
SELECT * FROM i18n_messages;
```

6. Drop the demo objects to prepare for another test. Connect to the database and run the txclean.sql script as follows:

```
txclean
```

7. Compile the Java demo program. For example:

```
javac txdemo1.java
```

8. Run the Java program, using the same JDBC and database connection data that you used when invoking the command-line interface. For example:

```
java txdemo1 "jdbc:oracle:thin:@//myhost:1521/myservicename" user -pw\txdemo1.xml
```

Perform the same query test (step 5) and clean-up operation (step 6) as before.

9. Run the TransX Utility to unload data into the predefined XML format. For example:

```
transx -s "jdbc:oracle:thin:@//myhost:1521/myservicename" user -pw emp.xml emp
```

Compare the data in emp.xml with emp-dlf.xml.

**Note:** For simplicity in demonstrating this feature, this example does not perform the password management techniques that a deployed system normally uses. In a production environment, follow the Oracle Database password management guidelines, and disable any sample accounts. See Oracle Database Security Guide for password management guidelines and other security recommendations.
Using the TransX Command-Line Utility

TransX utility is packaged with Oracle Database. By default, the Oracle Universal Installer installs the utility on disk. As explained in "Java XDK Component Dependencies" on page 3-2, the TransX library is $ORACLE_HOME/bin/xml.jar (UNIX) and %ORACLE_HOME%in\xml.jar (Windows).

You can run the TransX utility from the operating system command line with the following syntax:

```
java oracle.xml.transx.loader
```

The XDK includes a script version of TransX named $ORACLE_HOME/bin/transx (UNIX) and %ORACLE_HOME%in\transx.bat (Windows). Assuming that your PATH variable is set correctly, you can run TransX as follows:

```
transx options parameters
transx.bat options parameters
```

For example, the following command shows valid syntax:

```
transx -s 'jdbc:oracle:thin:@//myhost:1521/myservicename' user -pw emp.xml emp
```

TransX utility Command-Line Options

Table 12–4 describes the options for the TransX utility.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-u</td>
<td>Update existing rows.</td>
<td>Does not skip existing rows but updates them. To exclude a column from the update operation, set the useforupdate attribute to no.</td>
</tr>
<tr>
<td>-e</td>
<td>Raise exception if a row is already existing in the database.</td>
<td>Throws an exception if a duplicate row is found. By default, TransX skips duplicate rows. Rows are considered duplicate if the values for lookup-key column(s) in the database and the data set are the same.</td>
</tr>
<tr>
<td>-x</td>
<td>Print data in the database in the predefined format.</td>
<td>Similar to the -s option, it causes the utility to perform the opposite operation of loading. Unlike the -s option, it prints to stdout. Redirecting this output to a file is discouraged because intervention of the operating system may result in data loss due to unexpected transcoding.</td>
</tr>
<tr>
<td>-s</td>
<td>Save data in the database into a file in the predefined format.</td>
<td>Performs unloading. TransX Utility queries the database, formats the result into the predefined XML format, and stores it under the specified file name.</td>
</tr>
<tr>
<td>-p</td>
<td>Print the XML to load.</td>
<td>Prints out the data set for insert in the canonical format of XSU.</td>
</tr>
<tr>
<td>-t</td>
<td>Print the XML for update.</td>
<td>Prints out the data set for update in the canonical format of XSU.</td>
</tr>
<tr>
<td>-o</td>
<td>Omit validation (as the data set is parsed it is validated by default).</td>
<td>Causes TransX Utility to skip the format validation, which is performed by default.</td>
</tr>
<tr>
<td>-v</td>
<td>Validate the data format and exit without loading.</td>
<td>Causes TransX Utility to perform validation and exit.</td>
</tr>
</tbody>
</table>
Note the following command-line option exceptions:

- -u and -e are mutually exclusive.
- -v must be the only option followed by data, as shown in the examples.
- -x must be the only option followed by connect information and a SQL query, as shown in the examples.

Omitting all arguments results in the display of the usage information shown in Table 12–4.

### TransX Utility Command-Line Parameters

Table 12–5 describes the command-line parameters for the TransX utility.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect_string</td>
<td>The JDBC connect string. See Oracle Database JDBC Developer’s Guide and Reference,</td>
</tr>
<tr>
<td>username</td>
<td>Database user name.</td>
</tr>
<tr>
<td>password</td>
<td>Password for the database user, or &quot;-pw&quot;.</td>
</tr>
<tr>
<td>datasource</td>
<td>An XML document specified by filename or URL.</td>
</tr>
<tr>
<td>options</td>
<td>Described in Table 12–4, “TransX utility Command-line Options”.</td>
</tr>
</tbody>
</table>

**See Also:** Oracle Database XML Java API Reference for complete details of the TransX interface.

### Loading Data with the TransX Utility

The TransX utility is especially useful for populating a database with multilingual data. To use the utility to transfer data in and out of a database schema you must create a dataset that maps to this schema. This section describes a typical use scenario in which you use TransX to organize translated application messages in a database.

This section contains the following topics:

- Storing Messages in the Database
- Creating a Dataset in a Predefined Format
- Loading the Data
Storing Messages in the Database

To build an internationalized system, it is essential to decouple localizable resources from business logic. A typical example of such a resource is translated text information. Data that is specific to a particular region and shares a common language and cultural conventions needs to be organized with a resource management facility that can retrieve locale-specific information. A database is often used to store such data because of easy maintenance and flexibility.

Assume that you create the table with the structure and content shown in Example 12–1 and insert data.

Example 12–1 Structure of Table translated_messages

```sql
CREATE TABLE translated_messages
(
    MESSAGE_ID       NUMBER(4)
    CONSTRAINT tm_mess_id_nn NOT NULL,
    LANGUAGE_ID      VARCHAR2(42),
    MESSAGE          VARCHAR2(200)
);
```

The column `language_id` is defined in this table so that applications can retrieve messages based on the preferred language of the end user. It contains abbreviations of language names to identify the language of messages.

Example 12–2 shows sample data for the table.

Example 12–2 Query of translated_messages

```
MESSAGE_ID  LANGUAGE_ID  MESSAGE
----------  -----------  ----------------------------------
1           us           Welcome to System X
2           us           Please enter username and password
```

See Also: Oracle Database Globalization Support Guide for Oracle language abbreviations

Creating a Dataset in a Predefined Format

Chapter 13, "Data Loading Format (DLF) Specification" describes the complete syntax of the Data Loading Format (DLF) language. This language is used to create a DLF document that provides the input to TransX.

Given the dataset (the input data) in the canonical format, the TransX utility loads the data into the designated locations in the database. Note that TransX does not create the database objects: you must create the tables or views before attempting to load data.

An XML document that represents the `translated_messages` table created in Example 12–1 looks something like Example 12–3. The dataset reflects the structure of the target table, which in this case is called `translated_messages`.

Example 12–3 example.xml

```
<?xml version="1.0"?>
<table name="translated_messages">
    <!-- Specify the unique identifier -->
</table>
```
<lookup-key>
  <column name="message_id" />
  <column name="language_id" />
</lookup-key>

<!-- Specify the columns into which data will be inserted -->
<columns>
  <column name="message_id" type="number"/>
  <column name="language_id" type="string" constant="us" translate="yes"/>
  <column name="message" type="string" translate="yes"/>
</columns>

<!-- Specify the data to be inserted -->
<dataset>
  <row>
    <col name="message_id">1</col>
    <col name="message" translation-note="dnt'X'">Welcome to System X</col>
  </row>
  <row>
    <col name="message_id">2</col>
    <col name="message">Please enter username and password</col>
  </row>
  <!-- ... -->
</dataset>
</table>

**Format of the Input XML Document**

The XML document in **Example 12–3** starts with the following declaration:

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

Its root element `<table>`, which has an attribute that specifies the name of the table, encloses all the other elements:

```xml
<table name="translated_messages">
  ...
</table>
```

As explained in "**Elements in DLF**" on page 13-5, the `<table>` element contains three subsections:

- **Lookup Key Elements**
- **Metadata Elements**
- **Data Elements**

The preceding sections map to elements in **Example 12–3** as follows:

```xml
<lookup-key>...</lookup-key>
<columns>...</columns>
<dataset>...</dataset>
```

The lookup keys are columns used to evaluate rows if they already exist in the database. Because we want a pair of message and language IDs to identify a unique string, the document lists the corresponding columns. Thus, the `message_id`, `language_id`, and `message` columns in table `translated_messages` map to the attributes in the `<column>` element as follows:

```xml
<column name="message_id" type="number"/>
<column name="language_id" type="string" constant="us" translate="yes"/>
<column name="message" type="string" translate="yes"/>
```
The columns section should mirror the table structure because it specifies which piece of data in the dataset section maps to which table column. The column names should be consistent throughout the XML dataset and database. You can use the `<column>` attributes in Table 12–6 to describe the data to be loaded. Note that these attributes form a subset of the DLF attributes described in "Attributes in DLF" on page 13-7.

### Table 12–6 `<column>` Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Specifies the datatype of a column in the dataset. This attribute specifies the kind of text contained in the <code>&lt;col&gt;</code> element in the dataset. Depending on this type, the data loading tool applies different datatype conventions to the data.</td>
<td><code>&lt;column name=&quot;col&quot; type=&quot;string&quot; /&gt;</code></td>
</tr>
<tr>
<td>constant</td>
<td>Specifies a constant value. A column with a fixed value for each row does not have to repeat that same value.</td>
<td><code>&lt;column name=&quot;col&quot; type=&quot;string&quot; constant=&quot;us&quot; /&gt;</code></td>
</tr>
<tr>
<td>language</td>
<td>The language attribute indicates that the column is the language column, which stores a language tag. It works in the same way as the constant attribute, except for the role to declare the column is the language column.</td>
<td><code>&lt;column name=&quot;language&quot; type=&quot;string&quot; language=&quot;us&quot; /&gt;</code></td>
</tr>
<tr>
<td>sequence</td>
<td>Specifies a sequence in the database used to fill in the value for this column.</td>
<td><code>&lt;column name=&quot;id&quot; type=&quot;number&quot; sequence=&quot;id_sq&quot; /&gt;</code></td>
</tr>
<tr>
<td>translate</td>
<td>Indicates whether the text of this column or parameter is to be translated.</td>
<td><code>&lt;column name=&quot;msg&quot; type=&quot;string&quot; translate=&quot;yes&quot; /&gt;</code></td>
</tr>
</tbody>
</table>

The `constant` attribute of a `<column>` element specifies a value to be stored into the corresponding column for every row in the dataset section. Because in this example we are working in the original language, the language_id column is set to the value `us`.

### Defining the Language Column

Alternatively, the language_id column may use the `language` attribute instead of the `constant` attribute. A DLF document with the `language` attribute can use the `lang` attribute in the xml namespace. A language column can use the "%x" placeholder to get its value from the standard `xml:lang` attribute at the root table element. Thus `translate="yes"` is not needed, because the value "%x" does not have to be translated. The result of loading this DLF is the same as Example 10-3.

### Example 12–4 example.xml with a Language Attribute

```xml
<?xml version="1.0"?>
<table xml:lang="us" name="translated_messages">
  <!-- Specify the unique identifier -->
  <lookup-key>
    <column name="message_id" />
    <column name="language_id" />
  </lookup-key>
  <!-- Specify the columns into which data will be inserted -->
  <columns>
    <column name="message_id" type="number"/>
    <column name="language" type="string" language="us"/>
    <!-- Additional columns here... -->
  </columns>
</table>
```
As explained in Table 13–10 on page 13-8, the valid values for the type attribute are string, number, date, and dateTime. These values correspond to the datatypes defined in the XML schema standard, so each piece of data should conform to the respective datatype definition. In particular, it is important to use the ISO 8601 format for the date and dateTime datatypes, as shown in Table 12–7.

### Table 12–7 date and dateTime Formats

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>CCYY-MM-DD</td>
<td>2009-05-20</td>
</tr>
<tr>
<td>dateTime</td>
<td>CCYY-MM-DDThh:mm:ss</td>
<td>2009-05-20T16:01:37</td>
</tr>
</tbody>
</table>

Example 12–5 shows how you can represent a table row with dateTime data in a TransX dataset.

**Example 12–5 dateTime Row**

```xml
<row>
   <col name="article_id">12345678</col>
   <col name="author_id">10500</col>
   <col name="submission">2002-03-09T16:01:37</col>
   <col name="title">...</col>
   <!-- some columns follows -->
</row>
```

Specifying Translations in a Dataset

As explained in "Attributes in DLF" on page 13-7, you can use the translation attribute to specify whether the column contains translated data. In Example 12–3, two column elements use the translate attribute differently. The attribute for the language_id column specifies that the value of the constant attribute should be translated:

```xml
<column name="language_id" type='string' constant='us' translate='yes'/>
```

In contrast, the following translate attribute requests translation of the data in the dataset section with a name that matches this column:

```xml
<column name="message" type='string' translate='yes'/>
```

For example, the preceding element specifies that the following messages in the dataset section should be translated:
When translating messages for applications, you may decide that specified words or phrases should be left untranslated. The translation-note attribute shown in the preceding example achieves this goal.

An XSLT processor can convert the preceding format into another format for exchanging translation data among localization service providers for use with XML-based translation tools. This transformation insulates developers from tasks such as keeping track of revisions, categorizing translatable strings into units, and so on.

Example 12–6 shows what the document in Example 12–3 looks like after translation.

**Example 12–6  example_es.xml**

```xml
<?xml version="1.0"?>
<table xml:lang="es" name="translated_messages">
  <!-- Specify the unique identifier -->
  <lookup-key>
    <column name="message_id" />
    <column name="language_id" />
  </lookup-key>
  <!-- Specify the columns into which data will be inserted -->
  <columns>
    <column name="message_id" type="number"/>
    <column name="language_id" type="string" constant="es" translate="yes"/>
  </columns>
</table>
```

Example 12–7  shows what the document in Example 12–4 looks like after translation. Unlike the previous example, the column definition is not changed.

**Example 12–7  example_es.xml with a Language Attribute**

```xml
<?xml version="1.0"?>
<table xml:lang="es" name="translated_messages">
  <!-- Specify the unique identifier -->
  <lookup-key>
    <column name="message_id" />
    <column name="language_id" />
  </lookup-key>
  <!-- Specify the columns into which data will be inserted -->
  <columns>
    <column name="message_id" type="number"/>
    <column name="language_id" type="string" language="%x"/>
  </columns>
</table>
```

If you use a text editor or a traditional text-based translation tool during the translation process, it is important to maintain the encoding of the document. After a document is translated, it may be in a different encoding from the original. As explained in "XML Declaration in DLF" on page 13-4, If the translated document is in an encoding other than Unicode, then add the encoding declaration to the XML declaration on the first line. A declaration for non-Unicode encoding looks like the following:

```xml
<?xml version="1.0" encoding="ISO-8859-15"?>
```
To ensure that the translation process does not lose syntactic integrity, process the document as XML. Otherwise, you can check the format by specifying the -v option of the command-line interface. If a syntactic error exists, the utility prints the location and description of the error. You must fix errors for the data transfer to succeed.

**See Also:** Chapter 13, "Data Loading Format (DLF) Specification"

### Loading the Data

Suppose that you want to load the sample documents in Example 12–3 and Example 12–8 into the translated_messages table that you created in Example 12–1. You can use the sample program in Example 12–8, which you can find in the TransX demo directory, to load the data.

#### Example 12–8  txdemo1.java

```
// Copyright (c) 2001 All rights reserved Oracle Corporation
import oracle.xml.transx.*;

public class txdemo1 {

/**
 * Constructor
 */
  public txdemo1() {
  }

/**
 * main
 * @param args
 *
 * args[0] : connect string
 * args[1] : username
 * args[2] : password
 * args[3+]: xml file names
 */
  public static void main(String[] args) throws Exception {

    // instantiate a transx class
    TransX  transx = loader.getLoader();

    // start a data loading session
    transx.open( args[0], args[1], args[2] );

    // specify operation modes
    transx.setLoadingMode( LoadingMode.SKIP_DUPLICATES );
    transx.setValidationMode( false );

    // load the dataset(s)
    for ( int i = 3 ; i < args.length ; i++ )
      {  
        transx.load{ args[i] };  
      }

    // cleanup
    transx.close();
    }
  }
```

See Also: Chapter 13, "Data Loading Format (DLF) Specification"
The txdemo1.java program follows these steps:

1. Create a TransX loader object. For example:
   ```java
   TransX transx = loader.getLoader();
   ```

2. Open a data loading session. The first three command-line parameters are the JDBC connect string, database username, and database password. These parameters are passed to the TransX.open() method. The program includes the following statement:
   ```java
   transx.open( args[0], args[1], args[2] );
   ```

3. Configure the TransX loader. The program configures the loader to skip duplicate rows and to validate the input dataset. The program includes the following statements:
   ```java
   transx.setLoadingMode( LoadingMode.SKIP_DUPLICATES );
   transx.setValidationMode( false );
   ```

4. Load the data. The first three command-line parameters specify connection information; any additional parameters specify input XML documents. The program invokes the load() method for every specified document:
   ```java
   for ( int i = 3 ; i < args.length ; i++ )
   {
       transx.load( args[i] );
   }
   ```

5. Close the data loading session. The program includes the following statement:
   ```java
   transx.close();
   ```

After compiling the program with javac, you can run it from the command line. The following example uses the Java thin driver to connect to instance mydb on port 1521 of computer myhost. It connects to the user schema and loads the XML documents example.xml and example_es.xml:

```
java txdemo1 'jdbc:oracle:thin:@//myhost:1521/mydb" user -pw example.xml example_es.xml
```

In building a multilingual software system, translations usually become available at a later stage of development. They also tend to evolve over a period of time. If you need to add messages to the database, then you can add new rows in your <dataset> definition by running the TransX utility again. TransX recognizes which rows are new and inserts only the new messages based on the columns specified in the <lookup-key> section. If some messages are updated, then run TransX with the -u option to update existing rows with the data specified in XML, as shown in the following example:

```
transx -u 'jdbc:oracle:thin:@//myhost:1521/mydb" user -pw example.xml example_es.xml
```

### Querying the Data

After using the program in Example 12–8 to load the data, you can query the translated_messages table to see the results. The results should look like the following:

<table>
<thead>
<tr>
<th>MESSAGE_ID</th>
<th>LANGUAGE_ID</th>
<th>MESSAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>us</td>
<td>Welcome to System X</td>
</tr>
</tbody>
</table>
An application can retrieve a message in a specific language by using the `language_id` and `message_id` columns in a `WHERE` clause. For example, you can execute the following query:

```sql
SELECT message
FROM translated_messages
WHERE message_id = 2
AND language_id = 'es';
```

MESSAGE
------------------------
Porfavor entre su nombre de usuario y su contraseña
This chapter describes version 1.0 of the Data Loading Format (DLF), which is the standard format to describe translated messages and seed data loaded into the database by the TransX utility. It contains the following topics:

- Introduction to DLF
- General Structure of DLF
- DLF Specifications
- DLF Examples
- DLF References

**See Also:** Chapter 12, "Using the TransX Utility"

### Introduction to DLF

DLF defines a standard format for loading data with the TransX utility. It is intended to supersede loading data with SQL scripts. DLF provides the following advantages:

- Format validation. Validation results in fewer errors during the translation and loading processes.
- Ease of use. The user does not have to maintain the character encoding of each data file to correspond with the language used in the data file.

DLF is based on the XML 1.0 specification.

### Naming Conventions for DLF

This section describes the naming conventions used in this document.

**Elements and Attributes**

The following naming conventions for elements and attributes are used throughout this specification:

- Standard English letters
- Lowercase letters only
- Hyphen (-) may be used for concatenation
- Attribute names must be consistently defined throughout
- Industry-standard terminology should be followed wherever possible
Values
Values are case-sensitive except for some attribute values used for column names. All predefined attribute values are lowercase. No element values are defined by this specification.

File Extensions
No file extension is recommended by this specification. A future version of this specification may recommend that documents use an extension that conforms to an 8.3 standard.

General Structure of DLF
Data Loading Format is XML, so it begins with an XML declaration. After the XML declaration comes the DLF document itself, enclosed within the `<table>` element. A DLF document is composed of the following required sections:

- The `<lookup-key>` element contains a list of column names that determine whether existing rows in the database are duplicates of the rows in the dataset definition included in the `<dataset>` element.
- The `<columns>` element contains metadata about the `<dataset>` such as the names, datatypes, and attributes of columns.
- The `<dataset>` element contains a `<row>` element for each row, which in turn contains a `<col>` element that corresponds to a piece of data that is loaded in a database column. In this way a DLF document looks similar to the familiar tabular format in printing data in the database and allows easy editing.

DLF provides one optional section, which is enclosed within a `<translation>` element. This section may precede the required sections.

In addition, DLF provides information about TransX utility processing. Such information includes but is not limited to the following:

- The `<query>` element is used to retrieve the value to be loaded to the column from a SQL query.
- The `sequence` attribute is used to retrieve the value to be loaded to the column from a sequence object in the database.
- The `constant` attribute is used to specify a constant value to the column.
- The `language` attribute is used to specify the language identifier to be loaded to the column.

Tree Structure of DLF
This section shows the possible structure of a DLF document as a tree. Each element is represented as `<element_name>`, where `element_name` is the name of an element. Attributes have no markup. Each element and attribute is followed by notation indicating its possible occurrence. Table 13–1 describes the occurrence notation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
</tr>
<tr>
<td>+</td>
<td>one or more</td>
</tr>
<tr>
<td>?</td>
<td>zero or one</td>
</tr>
</tbody>
</table>
Example 13–1 shows the tree structure of a DLF document. The elements are described in "Elements in DLF" on page 13-5. The attributes are described in "Attributes in DLF" on page 13-7.

**Example 13–1  DLF Tree Structure**

```xml
<table>
  | 1
  +---- lang?
  |
  +---- space?
  |
  +---- normalize-langtag?
  |
  +---- <translation>?
    | 1
    +---- <target>*
    |
    | 1
    +---- <language ID>
    |
    +---- <restype>*
    |
    | 1
    +---- name1
    |
    +---- expansion?
    |
  +---- <lookup-key>1
    | 1
    +---- <column>*
    |
    | 1
    +---- name1
    |
  +---- <columns>1
    | 1
    +---- <column>*
    |
    | 1
    +---- name1
    |
    +---- type1
    |
    +---- translate?
    |
    +---- translation-note?
    |
    +---- constant?
    |
    +---- language?
    |
    +---- sequence?
    |
    +---- virtual?
    |
    +---- useforupdate?
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>zero or more</td>
</tr>
<tr>
<td>(a</td>
<td>b</td>
</tr>
</tbody>
</table>

Table 13–1 (Cont.) Notation for Occurrence of Attributes and Elements
This section contains the following topics:

- XML Declaration in DLF
- Entity References in DLF
- Elements in DLF
- Attributes in DLF

XML Declaration in DLF

The XML declaration starts an XML entity. It indicates the XML version and can be used to declare the encoding of the file, as in the following example:
<?xml version="1.0" encoding="iso-8859-1" ?>

As in all XML files, the default encoding for a DLF file is assumed to be either UTF-8, which is a superset of the 7-bit ASCII character set, or UTF-16, which is conceptually UCS-2 with surrogate pairs for code points above 65,535. Thus, for these character sets, the encoding declaration is not necessary. Furthermore, all XML parsers support these character sets. If the encoding is UTF-16, then the first character of the file must be the Unicode Byte-Order-Mark, #xFEFF, which indicates the endianness of the file.

Other character sets supported by Oracle XML parsers include all Oracle character sets and commonly used IANA character set and Java encodings. The names of these character sets can be found in the parser documentation. You must declare these with encoding declarations if the document does not have an external source of encoding information such as from the execution environment or the network protocol. Therefore, it is recommended that you use a Unicode character encoding so that you can dispense with the encoding declaration. The recommended practice is to encode the document in UTF-8 and use the following declaration:

<?xml version="1.0" ?>

Entity References in DLF

There are five entity references predefined by XML. These entity references are listed in Table 13–2. The &lt; and &amp; entity references must always be used in place of the character they reference.

<table>
<thead>
<tr>
<th>Entity Reference</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than sign (&lt;)</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than sign (&gt;)</td>
</tr>
<tr>
<td>&amp;</td>
<td>Ampersand (&amp;)</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe or single quote (‘)</td>
</tr>
<tr>
<td>&quot;</td>
<td>Straight, double quotation mark (&quot;)</td>
</tr>
</tbody>
</table>

Elements in DLF

The DLF elements shown in Example 13–1 are divided into the categories described in Table 13–3. Attributes are shared among them. The attributes are described in "Attributes in DLF" on page 13-7.

<table>
<thead>
<tr>
<th>Type of Element</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top Level Table Element</td>
<td>&lt;table&gt;</td>
</tr>
<tr>
<td>Translation Elements</td>
<td>&lt;target&gt;, &lt;restype&gt;</td>
</tr>
<tr>
<td>Lookup Key Elements</td>
<td>&lt;lookup-key&gt;, &lt;column&gt;</td>
</tr>
<tr>
<td>Metadata Elements</td>
<td>&lt;columns&gt;, &lt;column&gt;, &lt;query&gt;, &lt;sql&gt;, &lt;parameter&gt;</td>
</tr>
<tr>
<td>Data Elements</td>
<td>&lt;dataset&gt;, &lt;row&gt;, &lt;col&gt;</td>
</tr>
</tbody>
</table>

Top Level Table Element

Table 13–10 describes the top level table element.
Translation Elements

Table 13–5 describes the translation elements.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Required Attributes</th>
<th>Optional Attributes</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;translation&gt;</td>
<td>Contains generic information pertinent to translation.</td>
<td>None</td>
<td>None</td>
<td>Zero or more &lt;target&gt; elements and zero or more &lt;restype&gt; elements</td>
</tr>
<tr>
<td>&lt;target&gt;</td>
<td>Specifies a language to which this document is translated.</td>
<td>None</td>
<td>None</td>
<td>A language identifier as defined by [IETF RFC 1766]</td>
</tr>
<tr>
<td>&lt;restype&gt;</td>
<td>Declares a type of resource.</td>
<td>name expansion</td>
<td>Empty element</td>
<td></td>
</tr>
</tbody>
</table>

Lookup Key Elements

Table 13–6 describes the lookup key elements.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Required Attributes</th>
<th>Optional Attributes</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;lookup-key&gt;</td>
<td>Contains the &lt;column&gt; element(s) based on which the TransX recognizes the rows as identical or duplicate.</td>
<td>name</td>
<td>None</td>
<td>Zero or more &lt;column&gt; elements</td>
</tr>
<tr>
<td>&lt;column&gt;</td>
<td>A &lt;column&gt; element under &lt;lookup-key&gt; element indicates a column to be used to recognize the rows as identical or duplicate. Columns with the same values in specified column(s) are considered duplicate, regardless of the values in the other column(s). A lookup key &lt;column&gt; must have corresponding &lt;columns&gt; in the &lt;dataset&gt; portion or be declared as a &lt;column&gt; with a constant expression or a &lt;query&gt; in the &lt;columns&gt; section.</td>
<td>name</td>
<td>None</td>
<td>Empty element</td>
</tr>
</tbody>
</table>

Metadata Elements

Table 13–7 describes the metadata elements.
Table 13–7  Metadata Elements

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Required Attributes</th>
<th>Optional Attributes</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;columns&gt;</td>
<td>Contains data about the data to be loaded.</td>
<td>None</td>
<td>None</td>
<td>One or more &lt;column&gt; elements</td>
</tr>
<tr>
<td>&lt;column&gt;</td>
<td>Specifies a column that corresponds to &lt;col&gt; elements under the &lt;dataset&gt; element. Once a &lt;column&gt; is defined the corresponding &lt;col&gt; element must appear in every &lt;row&gt; unless the column has the sequence, constant or query attribute.</td>
<td>name, type in either order. The recommended sequence is name, type, then optional attributes. translate, constant, sequence, virtual, useforupdate, maxsize, size-unit, restype in any order</td>
<td>Zero or one &lt;query&gt; or &lt;sql&gt; element</td>
<td></td>
</tr>
<tr>
<td>&lt;query&gt;</td>
<td>Specifies a SQL query whose result is used to fill in the column to which this element belongs.</td>
<td>text</td>
<td>None</td>
<td>Zero or more &lt;parameter&gt; elements</td>
</tr>
<tr>
<td>&lt;sql&gt;</td>
<td>Specifies a SQL statement whose result, if any, is used to fill in the column to which this element belongs.</td>
<td>text</td>
<td>None</td>
<td>Zero or more &lt;parameter&gt; elements</td>
</tr>
<tr>
<td>&lt;parameter&gt;</td>
<td>Specifies a parameter of a &lt;query&gt; element.</td>
<td>id and either col or constant. If col is specified, the referenced column cannot have the query, constant, or sequence attributes. translate, trans-key</td>
<td>Empty</td>
<td></td>
</tr>
</tbody>
</table>

Data Elements

Table 13–8 describes the data elements.

Table 13–8  Data Elements

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Required Attributes</th>
<th>Optional Attributes</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;dataset&gt;</td>
<td>Contains data to be loaded into the database.</td>
<td>None</td>
<td>None</td>
<td>One or more &lt;row&gt; elements</td>
</tr>
<tr>
<td>&lt;row&gt;</td>
<td>Contains data about the data to be loaded</td>
<td>None</td>
<td>None</td>
<td>Zero or more &lt;col&gt; elements</td>
</tr>
<tr>
<td>&lt;col&gt;</td>
<td>Specifies an instance of a piece of data to be loaded to a database column, or in the case of a virtual column, a piece of data to be used to obtain an actual data to be loaded to a database column.</td>
<td>name</td>
<td>trans-key</td>
<td>Data for use by applications</td>
</tr>
</tbody>
</table>

Attributes in DLF

This section lists the various attributes used in the DLF elements. An attribute is never specified more than once for each element. Along with some of the attributes are the Recommended Attribute Values. Values for these attributes are case sensitive.

Table 13–9  Attributes

<table>
<thead>
<tr>
<th>Type of Attribute</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLF Attributes</td>
<td>name, type, translate, constant, sequence, virtual, useforupdate, maxsize, size-unit, restype, text, id, col, trans-key, translation-note, normalize-langtag</td>
</tr>
<tr>
<td>XML Namespace Attributes</td>
<td>xml:space</td>
</tr>
</tbody>
</table>
**DLF Attributes**

Table 13–10 describes the DLF attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Value Description</th>
<th>Default Value</th>
<th>Used by Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>lang</td>
<td>Specifies the language of the document.</td>
<td>This is equivalent to the xml:lang attribute. The values of the attribute are language identifiers as defined by [IETF RFC 4646]. This attribute does not affect data loading operation in any way.</td>
<td>None; if absent, &quot;en&quot; is assumed</td>
<td>&lt;table&gt;</td>
</tr>
<tr>
<td>normalize-langtag</td>
<td>Specifies how to normalize the case of language tag.</td>
<td>&quot;none&quot;, &quot;standard&quot;, &quot;uppercase&quot; or &quot;lowercase&quot;. The meanings are as follows: none - no normalization, the values in the language column on DLF are used as they are, standard - the style as recommended by the standards * lowercase for the 2 letter language code * uppercase for the 2 letter country code * titlecase for the 4 letter script code * lowercase for others uppercase - uppercase everything lowercase - lowercase everything</td>
<td>none</td>
<td>&lt;table&gt;</td>
</tr>
<tr>
<td>space</td>
<td>Specifies how white spaces (ASCII spaces, tabs and line-breaks) should be treated.</td>
<td>&quot;default&quot; or &quot;preserve&quot; The value &quot;default&quot; signals that applications' default white-space processing modes are acceptable; the value &quot;preserve&quot; indicates the intent that applications preserve all whitespace. If this intent is declared at the root table element, it is considered to apply to all string data elements in the whole document. If declared at column level, it is considered to apply to the specified column of every row. If this attribute is declared in the dataset section, the intent applies to the immediate text data only. Declaration at the document or column level may be overridden with another instance of the space attribute.</td>
<td>&quot;default&quot;</td>
<td>&lt;table&gt;, &lt;column&gt;, &lt;col&gt;</td>
</tr>
<tr>
<td>name</td>
<td>Specifies the name of an object such as table, column, restype, and so forth.</td>
<td>String: This is a database table name for the <code>&lt;table&gt;</code> element, and a column name for the <code>&lt;column&gt;</code> or <code>&lt;col&gt;</code> element.</td>
<td>n/a</td>
<td>&lt;table&gt;, &lt;column&gt;, &lt;col&gt;</td>
</tr>
</tbody>
</table>
Data Loading Format (DLF) Specification

The datatype of a column in the dataset. This attribute specifies the kind of text contained in the <col> element in the dataset. Depending on this type, TransX may apply different processes to the data.

Because implicit datatype conversion is provided by XSU and JDBC, TransX does not do its own parsing based on this type information. It uses this attribute to choose appropriate intermediate data types in Java for columns of date or dateTime type, in which case the standard date formats are accepted.

- **type**
  - String: possible values are "number", "string", "date", "dateTime" or "binary".
  - The lexical representation of a value of number type should be supplied in the SQL language syntax, no matter what the current locale is. The SQL syntax uses no digit grouping separator (usually comma), but uses a dot as the decimal separator (usually dot).
  - For the binary data type, the data value specified in a text field between the col tags indicates the name of a file that contains the actual binary data. Raw data cannot be embedded in the text field.
  - For the other data types (string, date, and dateTime) the representation is constrained by the corresponding types in the XML Schema specification.
  - For the sake of simplicity, DLF only accepts standard date formats of XML Schema in the form "CCYY-MM-DDThh:mm:ss" (dateTime) or "CCYY-MM-DD" (date). No other date format is recognized.
  - TransX uses this attribute for the following:
    - Bind virtual columns to parameters of a query
    - Bind the result of a query to a corresponding column

- **translate**
  - Indicates whether or not the text of this column or parameter should be translated.
  - Either "yes" or "no"
  - "no"
  - <column>, <parameter>

- **constant**
  - Specifies a constant value for this column or parameter.
  - The value of this column for every row
  - n/a
  - <column>, <parameter>

- **language**
  - Specifies language identifier for this column
  - Language identifier or a placeholder. "%x" gets the value from the xml:lang attribute of the root table element.
  - n/a
  - <column>

- **sequence**
  - Specifies a sequence in the database used to fill in the value for this column.
  - String: The name of a sequence in the database
  - n/a
  - <column>

- **virtual**
  - Indicates that this column provides data used to construct another piece of data, which in turn is loaded into the database.
  - A virtual column does not exist in the database. It is typically used to provide a value of a parameter in a query. A virtual column cannot be a lookup-key column. A virtual column with a query throws the result away.
  - Either "yes" or "no"
  - "no"
  - <column>

- **useforupdate**
  - Indicates whether the value of this column is used for the update when uploading.seed data. This attribute has no effect unless TransX is in the mode to update duplicate rows. A virtual column cannot have this attribute set to "yes".
  - Either "yes" or "no".
  - If this attribute is set to "no", then the value of the column remains unchanged on the update operation.
  - "yes"
  - <column>

### Table 13–10 (Cont.) DLF Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Value Description</th>
<th>Default Value</th>
<th>Used by Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The datatype of a column in the dataset. This attribute specifies the kind of text contained in the &lt;col&gt; element in the dataset. Depending on this type, TransX may apply different processes to the data. Because implicit datatype conversion is provided by XSU and JDBC, TransX does not do its own parsing based on this type information. It uses this attribute to choose appropriate intermediate data types in Java for columns of date or dateTime type, in which case the standard date formats are accepted.</td>
<td>String: possible values are &quot;number&quot;, &quot;string&quot;, &quot;date&quot;, &quot;dateTime&quot; or &quot;binary&quot;. The lexical representation of a value of number type should be supplied in the SQL language syntax, no matter what the current locale is. The SQL syntax uses no digit grouping separator (usually comma), but uses a dot as the decimal separator (usually dot). For the binary data type, the data value specified in a text field between the col tags indicates the name of a file that contains the actual binary data. Raw data cannot be embedded in the text field. For the other data types (string, date, and dateTime) the representation is constrained by the corresponding types in the XML Schema specification. For the sake of simplicity, DLF only accepts standard date formats of XML Schema in the form &quot;CCYY-MM-DDThh:mm:ss&quot; (dateTime) or &quot;CCYY-MM-DD&quot; (date). No other date format is recognized. TransX uses this attribute for the following: - Bind virtual columns to parameters of a query - Bind the result of a query to a corresponding column</td>
<td>&quot;no&quot;</td>
<td>&lt;column&gt;</td>
</tr>
<tr>
<td>translate</td>
<td>Indicates whether or not the text of this column or parameter should be translated.</td>
<td>Either &quot;yes&quot; or &quot;no&quot;</td>
<td>&quot;no&quot;</td>
<td>&lt;column&gt;, &lt;parameter&gt;</td>
</tr>
<tr>
<td>constant</td>
<td>Specifies a constant value for this column or parameter.</td>
<td>The value of this column for every row</td>
<td>n/a</td>
<td>&lt;column&gt;, &lt;parameter&gt;</td>
</tr>
<tr>
<td>language</td>
<td>Specifies language identifier for this column</td>
<td>Language identifier or a placeholder. &quot;%x&quot; gets the value from the xml:lang attribute of the root table element.</td>
<td>n/a</td>
<td>&lt;column&gt;</td>
</tr>
<tr>
<td>sequence</td>
<td>Specifies a sequence in the database used to fill in the value for this column.</td>
<td>String: The name of a sequence in the database</td>
<td>n/a</td>
<td>&lt;column&gt;</td>
</tr>
<tr>
<td>virtual</td>
<td>Indicates that this column provides data used to construct another piece of data, which in turn is loaded into the database. A virtual column does not exist in the database. It is typically used to provide a value of a parameter in a query. A virtual column cannot be a lookup-key column. A virtual column with a query throws the result away.</td>
<td>Either &quot;yes&quot; or &quot;no&quot;</td>
<td>&quot;no&quot;</td>
<td>&lt;column&gt;</td>
</tr>
<tr>
<td>useforupdate</td>
<td>Indicates whether the value of this column is used for the update when uploading seed data. This attribute has no effect unless TransX is in the mode to update duplicate rows. A virtual column cannot have this attribute set to &quot;yes&quot;.</td>
<td>Either &quot;yes&quot; or &quot;no&quot;. If this attribute is set to &quot;no&quot;, then the value of the column remains unchanged on the update operation.</td>
<td>&quot;yes&quot;</td>
<td>&lt;column&gt;</td>
</tr>
</tbody>
</table>
Table 13–10 (Cont.) DLF Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Value Description</th>
<th>Default Value</th>
<th>Used by Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxsize</td>
<td>Specifies the maximum size for the data for this column.</td>
<td>Numeric value in the unit specified by the size-unit attribute. If this attribute is set and the size-unit is not set, the size is assumed to be in characters.</td>
<td>None</td>
<td>&lt;column&gt;</td>
</tr>
<tr>
<td>size-unit</td>
<td>Specifies the unit of size specified in the maxsize attribute.</td>
<td>Units. Recommended values are &quot;char&quot; for characters, &quot;byte&quot; for bytes. For supplemental characters, they take two &quot;char&quot; units.</td>
<td>&quot;char&quot;</td>
<td>&lt;column&gt;</td>
</tr>
<tr>
<td>retype</td>
<td>Indicates the type of data contained in this column. A resource type. The value must match with the name of a &lt;retype&gt; element.</td>
<td>None</td>
<td>&lt;column&gt;</td>
<td></td>
</tr>
<tr>
<td>expansion</td>
<td>Indicates the maximum size up to which translated strings are allowed to become longer for this type of resource.</td>
<td>A numeric value in percentage of increase.</td>
<td>0%</td>
<td>&lt;retype&gt;</td>
</tr>
<tr>
<td>text</td>
<td>Specifies a SQL query statement to obtain a value to put in the column to which the query belongs.</td>
<td>A SQL statement. Zero or more parameters can be specified with an identifier preceded by a colon. The statement should return a single row of a single value. Any excessive result is discarded.</td>
<td>n/a</td>
<td>&lt;query&gt;</td>
</tr>
<tr>
<td>id</td>
<td>Specifies a placeholder used in a SQL query statement with parameters. The value of the column specified by the sibling col attribute is associated as a parameter to the query.</td>
<td>String: an identifier that appears in the text attribute of the parent query element.</td>
<td>Empty string</td>
<td>&lt;parameter&gt;</td>
</tr>
<tr>
<td>col</td>
<td>Specifies a column to be associated with a placeholder in the query specified by the sibling id attribute.</td>
<td>String: a column name. The column must be other than the column this attribute is a part of.</td>
<td>n/a</td>
<td>&lt;parameter&gt;</td>
</tr>
<tr>
<td>trans-key</td>
<td>Specifies a key for translation.</td>
<td>String: a translation key. The value has to be unique in a translation domain.</td>
<td>n/a</td>
<td>&lt;col&gt;, &lt;parameter&gt;</td>
</tr>
<tr>
<td>translation-note</td>
<td>Specifies notes for translation.</td>
<td>String: Translation notes.</td>
<td>n/a</td>
<td>&lt;col&gt;, &lt;column&gt;, &lt;parameter&gt;</td>
</tr>
</tbody>
</table>

XML Namespace Attributes

Table 13–11 describes the XML namespace attributes.

Table 13–11 XML Namespace Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Value Description</th>
<th>Default Value</th>
<th>Used by Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml:space</td>
<td>Specifies how whitespace (ASCII spaces, tabs and line-breaks) should be treated.</td>
<td>&quot;default&quot; or &quot;preserve&quot; The value &quot;default&quot; signals that applications' default whitespace processing modes are acceptable for this element; the value &quot;preserve&quot; indicates the intent that applications preserve all the whitespace. This declared intent is considered to apply to all elements within the content of the element where it is specified, unless overridden with another instance of the xml:space attribute.</td>
<td>&quot;default&quot;</td>
<td>None</td>
</tr>
<tr>
<td>xml:lang</td>
<td>Specifies the language of the content.</td>
<td>A language tag defined by RFC 4646.</td>
<td>n/a</td>
<td>table</td>
</tr>
</tbody>
</table>
DLF Examples

This section contains the following topics:

- Minimal DLF Document
- Typical DLF Document
- Localized DLF Document

Minimal DLF Document

Example 13–2 shows a minimal DLF document.

Example 13–2 Minimal DLF Document

```xml
<?xml version="1.0" ?>
<table name="dual">
  <lookup-key/>
  <columns>
    <column name="DUMMY" type="string">
  </columns>
  <dataset>
    <row>
      <col name="DUMMY">X</col>
    </row>
  </dataset>
</table>
```

Typical DLF Document

Example 13–3 shows a sample DLF document that contains seed data for the CLK_STATUS_L table.

Example 13–3 Sample DLF Document

```xml
<?xml version="1.0" standalone="yes" encoding="UTF-8"?>
<table name="clk_status_l" xml:space="preserve">
  <lookup-key>
    <!--column name="status_id" /-->
    <column name="status_code" />
  </lookup-key>
</table>
```
<columns>
  <column name="status_id" type="number" sequence="clk_status_seq" useforupdate="no"/>
  <column name="status_code" type="number" />
  <column name="status_name" type="string" translate="yes" />
  <column name="status_description" type="string" translate="yes" />
  <column name="version_created" type="number" constant="0" />
  <column name="version_updated" type="number" constant="0" />
  <column name="status_type_code" type="string" virtual="yes" />
  <column name="status_type_id" type="number" >
    <query text="select status_type_id from clk_status_type_l where status_type_code = :1" >
      <parameter id="1" col="status_type_code" />
    </query>
  </column>
</columns>

<dataset>
  <row>
    <col name="status_code" >100</col>
    <col name="status_name" trans-key="stts-name-1" >Continue</col>
    <col name="status_description" trans-key="stts-desc-1" >The client should continue with its request.</col>
    <col name="status_type_code" >INFO</col>
  </row>
  <row>
    <col name="status_code" >101</col>
    <col name="status_name" trans-key="stts-name-2" >Switching Protocols</col>
    <col name="status_description" trans-key="stts-desc-2" >The server understands and is willing to comply with the client's request (via the Upgrade message header field) for a change in the application protocol being used on this connection.</col>
    <col name="status_type_code" >INFO</col>
  </row>
  <row>
    <col name="status_code" >200</col>
    <col name="status_name" trans-key="stts-name-3" >OK</col>
    <col name="status_description" trans-key="stts-desc-3" >The request has succeeded.</col>
    <col name="status_type_code" >SUCCESS</col>
  </row>
  <row>
    <col name="status_code" >201</col>
    <col name="status_name" trans-key="stts-name-4" >Created</col>
    <col name="status_description" trans-key="stts-desc-4" >The request has been fulfilled and resulted in a new resource being created.</col>
    <col name="status_type_code" >SUCCESS</col>
  </row>
  <row>
    <col name="status_code" >202</col>
    <col name="status_name" trans-key="stts-name-5" >Accepted</col>
    <col name="status_description" trans-key="stts-desc-5" >The request has been accepted for processing, but the processing has not been completed.</col>
    <col name="status_type_code" >SUCCESS</col>
  </row>
</dataset>
Localized DLF Document

**Example 13-4** shows an example of elements and attributes for localization.

**Example 13-4  DLF with Localization**

```xml
<?xml version="1.0"?>
<table name="table_name" xml:lang="en" xml:space="preserve">

<translation>
<target>ar</target>
<target>bs</target>
<target>es</target>
<restype name="alt" expansion="50%"/>
<restype name="foo" expansion="50%"/>
<restype name="bar" expansion="30%"/>
</translation>

<lookup-key><column name="resid" /></lookup-key>

<columns>
<column name="resid" type="number" sequence="seq_foo" useforupdate="no"/>
<column name="image" type="binary"/>
<column name="alt_text" type="string" translate="yes" maxsize="30"
size-unit="byte" restype="alt"/>
</columns>

<dataset>
<col name="image">foo1.gif</col>
<col name="alt_text">Hello world</col>
</dataset>
</table>
```
DLF References

See the following references for further information:

[IETFRFC4646] Tags for the Identification of Languages
http://www.ietf.org/rfc/rfc4646
This chapter contains these topics:

- Introduction to the XSQL Pages Publishing Framework
- Using the XSQL Pages Publishing Framework: Overview
- Generating and Transforming XML with XSQL Servlet
- Using XSQL in Java Programs
- XSQL Pages Tips and Techniques

### Introduction to the XSQL Pages Publishing Framework

The Oracle XSQL pages publishing framework is an extensible platform for publishing XML in multiple formats. The Java-based XSQL servlet, which is the center of the framework, provides a declarative interface for dynamically publishing dynamic Web content based on relational data.

The XSQL framework combines the power of SQL, XML, and XSLT. You can use it to create declarative templates called XSQL pages to perform the following actions:

- Assemble dynamic XML datagrams based on parameterized SQL queries
- Transform datagrams with XSLT to generate a result in an XML, HTML, or text-based format

An XSQL page, so called because its default extension is `.xsql`, is an XML file that contains instructions for the XSQL servlet. The Example 14–1 shows a simple XSQL page. It uses the `<xsql:query>` action element to query the `hr.employees` table.

#### Example 14–1  Sample XSQL Page

```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="emplist.xsl"?>
<xsql:query connection="hr" xmlns:xsql="urn:oracle-xsql"
  SELECT * FROM employees
</xsql:query>
```

You can present a browser client with the data returned from the query in Example 14–1. Assembling and transforming information for publishing requires no programming. You can perform most tasks in a declarative way. If one of the built-in features does not fit your needs, however, then you can use Java to integrate custom data sources or perform customized server-side processing.

In the XSQL pages framework, the assembly of information to be published is separate from presentation. This architectural feature enables you to do the following:
Using the XSQL Pages Publishing Framework: Overview

Prerequisites

This chapter assumes that you are familiar with the following technologies:

- **Oracle Database SQL.** The XSQL framework accesses data in a database.
- **PL/SQL.** The XDK supplies a PL/SQL API for XSU that mirrors the Java API.
- **Java Database Connectivity (JDBC).** The XSQL pages framework depends on a JDBC driver for database connections.
- **eXtensible Stylesheet Language Transformation (XSLT).** You can use XSLT to transform the data into a format appropriate for delivery to the user.
- **XML SQL Utility (XSU).** The XSQL pages framework uses XSU to query the database.

Using the XSQL Pages Publishing Framework: Overview

This section contains the following topics:

- **Using the XSQL Pages Framework: Basic Process**
- **Setting Up the XSQL Pages Framework**
- **Running the XSQL Pages Demo Programs**
- **Using the XSQL Pages Command-Line Utility**

Using the XSQL Pages Framework: Basic Process

The XSQL page processor engine interprets, caches, and processes the contents of XSQL pages. Figure 14–1 shows the basic architecture of the XSQL pages publishing framework. The XSQL page processor provides access from the following entry points:

- From the command line or in batch mode with the XSQL command-line utility. The `oracle.xml.xsql.XSQLCommandLine` class is the command-line interface.
- Over the Web by using the XSQL servlet installed in a Web server. The `oracle.xml.xsql.XSQLServlet` class is the servlet interface.
- As part of JSP applications by using `<jsp:include>` to include a template or `<jsp:forward>` to forward a template.
- Programatically by using the `oracle.xml.xsql.XSQLRequest` Java class.
You can run the same XSQL pages from any of the access points shown in Figure 14–1. Regardless of which way you use the XSQL page processor, it performs the following actions to generate a result:

1. Receives a request to process an XSQL page. The request can come from the command line utility or programmatically from an XSQLRequest object.

2. Assembles an XML datagram by using the result of one or more SQL queries. The query is specified in the <xsql:query> element of the XSQL page.

3. Returns this XML datagram to the requestor.

4. Optionally transforms the datagram into any XML, HTML, or text-based format.

Figure 14–2 shows a typical Web-based scenario in which a Web server receives an HTTP request for Page.xsql, which contains a reference to the XSLT stylesheet Style.xsl. The XSQL page contains a database query.
The XSQL page processor shown in Figure 14–2 performs the following steps:

1. Receives a request from the XSQL Servlet to process Page.xsql.
2. Parses Page.xsql with the Oracle XML Parser and caches it.
3. Connects to the database based on the value of the connection attribute on the document element.
4. Generates the XML datagram by replacing each XSQL action element, for example, <xsql:query>, with the XML results returned by its built-in action handler.
5. Parses the Style.xsl stylesheet and caches it.
6. Transforms the datagram by passing it and the Style.xsl stylesheet to the Oracle XSLT processor.
7. Returns the resulting XML or HTML document to the requester.

During the transformation step in this process, you can use stylesheets that conform to the W3C XSLT 1.0 or 2.0 standard to transform the assembled datagram into document formats such as the following:

- HTML for browser display
- Wireless Markup Language (WML) for wireless devices
- Scalable Vector Graphics (SVG) for data-driven charts, graphs, and diagrams
- XML Stylesheet Formatting Objects (XSL-FO), for rendering into Adobe PDF
Text documents such as e-mails, SQL scripts, Java programs, and so on
Arbitrary XML-based document formats

Setting Up the XSQL Pages Framework
You can develop and use XSQL pages in various scenarios. This section describes the following topics:
- Creating and Testing XSQL Pages with Oracle JDeveloper
- Setting the CLASSPATH for XSQL Pages
- Configuring the XSQL Servlet Container
- Setting Up the Connection Definitions

Creating and Testing XSQL Pages with Oracle JDeveloper
The easiest way to use XSQL pages is with Oracle JDeveloper 10g. The IDE supports the following features:
- Color-coded syntax highlighting
- XML syntax checking
- In-context drop-down lists that help you pick valid XSQL tag names and auto-complete tag and attribute names
- XSQL page deployment and testing
- Debugging tools
- Wizards for creating XSQL actions

To create an XSQL page in an Oracle JDeveloper 10g project, do the following steps:
1. Create or open a project.
2. Select File and then New.
3. In the New Gallery dialog box, select the General category and then XML.
4. In the Item window, select XSQL Page and click OK. JDeveloper loads a tab for the new XSQL page into the central window.

To add XSQL action elements such as <xsql:query> to your XSQL page, place the cursor where you want the new element to go and click an item in the Component Palette. A wizard opens that takes you through the steps of selecting which XSQL action you want to use and which attributes you need to provide.

To check the syntax of an XSQL page, place the cursor in the page and right-click Check XML Syntax. If there are any XML syntax errors, JDeveloper displays them.

To test an XSQL page, select the page in the navigator and right-click Run. JDeveloper automatically starts up a local Web server, properly configured to run XSQL pages, and tests your page by launching your default browser with the appropriate URL to request the page. After you have run the XSQL page, you can continue to make modifications to it in the IDE as well as to any XSLT stylesheets with which it might be associated. After saving the files in the IDE you can immediately refresh the browser to observe the effect of the changes.

You must add the XSQL runtime library to your project library list so that the CLASSPATH is properly set. The IDE adds this entry automatically when you go through the New Gallery dialog to create a new XSQL page, but you can also add it manually to the project as follows:
1. Right-click the project in the Applications Navigator.
2. Select **Project Properties**.
3. Select **Profiles** and then **Libraries** from the navigation tree.
4. Move **XSQL Runtime** from the **Available Libraries** pane to **Selected Libraries**.

**Setting the CLASSPATH for XSQL Pages**

Outside of the JDeveloper environment, you should make sure that the XSQL page processor engine is properly configured.

Make sure that the appropriate JAR files are in the **CLASSPATH** of the JavaVM that processes the XSQL Pages. The complete set of XDK JAR files is described in Table 3–1, "Java Libraries for XDK Components" on page 3-3. The JAR files for the XSQL framework include the following:

- `xml.jar`, the XSQL page processor
- `xmlparserv2.jar`, the Oracle XML parser
- `xsu12.jar`, the Oracle XML SQL utility (XSU)
- `ojdbc5.jar`, the Oracle JDBC driver (or `ojdbc6.jar`)

**Note:** The XSQL servlet can connect to any database that has JDBC support. Indicate the appropriate JDBC driver class and connection URL in the XSQL configuration file connection definition. Object-relational functionality only works when using Oracle database with the Oracle JDBC driver.

If you have configured your CLASSPATH as instructed in "Setting Up the Java XDK Environment" on page 3-5, then you only need to add the directory where the XSQL pages configuration file resides. In the database installation of the XDK, the directory for `XSQLConfig.xml` is `$ORACLE_HOME/xdk/admin`.

On Windows your `%CLASSPATH%` variable should contain the following entries:

```
%ORACLE_HOME%\lib\ojdbc5.jar;%ORACLE_HOME%\lib\xmlparserv2.jar;
%ORACLE_HOME%\lib\xsu12.jar;C:\xsql\lib\xml.jar;%ORACLE_HOME%\xdk\admin
```

On UNIX the `$CLASSPATH` variable should contain the following entries:

```
$ORACLE_HOME/lib/ojdbc5.jar:$ORACLE_HOME/lib/xmlparserv2.jar:
$ORACLE_HOME/lib/xsu12.jar:$ORACLE_HOME/lib/xml.jar:$ORACLE_HOME/xdk/admin
```

**Note:** If you are deploying your XSQL pages in a J2EE WAR file, then you can include the XSQL JAR files in the `.WEB-INF/lib` directory of the WAR file.

**Configuring the XSQL Servlet Container**

You can install the XSQL servlet in a variety of different Web servers, including OC4J, Jakarta Tomcat, and so forth. You can find complete instructions for installing the servlet in the Release Notes for the OTN download of the XDK.

Navigate to the setup instructions as follows:

1. Log on to OTN and navigate to the following URL:
Using the XSQL Pages Publishing Framework: Overview

http://www.oracle.com/technology/tech/xml/xdk/doc/production10g/readme.html

2. Click Getting Started with XDK Java Components.

3. In the Introduction section, scroll down to XSQL Servlet in the bulleted list and click Release Notes.

4. In the Contents section, click Downloading and Installing the XSQL Servlet.

5. Scroll down to the Setting Up Your Servlet Engine to Run XSQL Pages section and look for your Web server.

Setting Up the Connection Definitions

XSQL pages specify database connections by using a short name for a connection that is defined in the XSQL configuration file, which by default is named $ORACLE_HOME/xdk/admin/XSQLConfig.xml.

Note: If you are deploying your XSQL pages in a J2EE WAR file, then you can place the XSQLConfig.xml file in the .%/WEB-INF/classes directory of your WAR file.

The sample XSQL page shown in Example 14–1 contains the following connection information:

<xsql:query connection="hr" xmlns:xsql="urn:oracle-xsql">

Connection names are defined in the <connectiondefs> section of the XSQL configuration file. Example 14–2 shows the relevant section of the sample configuration file included with the database, with the hr connection in bold.

Example 14–2  Connection Definitions Section of XSQLConfig.xml

<connectiondefs>
  ...
  <connection name="hr">
    <username>hr</username>
    <password>hr_password</password>
    <dburl>jdbc:oracle:thin:@localhost:1521:ORCL</dburl>
    <driver>oracle.jdbc.driver.OracleDriver</driver>
    <autocommit>false</autocommit>
  </connection>
  ...
</connectiondefs>

For each database connection, you can specify the following elements:

- <username>, the database username
- <password>, the database password
- <dburl>, the JDBC connection string
- <driver>, the fully-qualified class name of the JDBC driver to use
- <autocommit>, which optionally forces AUTOCOMMIT to TRUE or FALSE

Specify an <autocommit> child element to control the setting of the JDBC autocommit for any connection. If no <autocommit> child element is set for a <connection>, then the autocommit setting is not set by the XSQL connection manager. In this case, the setting is the default autocommit setting for the JDBC driver.
You can place an arbitrary number of <connection> elements in the XSQL configuration file to define your database connections. An individual XSQL page refers to the connection it wants to use by putting a connection="xxx" attribute on the top-level element in the page (also called the "document element").

**Caution:** The XSQLConfig.xml file contains sensitive database username and password information that you want to keep secure on the database server. Refer to "Security Considerations for XSQL Pages" on page 14-29 for instructions.

### Running the XSQL Pages Demo Programs

Demo programs for the XSQL servlet are included in $ORACLE_HOME/xdk/demo/java/xsql. Table 14–1 lists the demo subdirectories and explains the included demos. The Demo Name column refers to the title of the demo listed on the XSQL Pages & XSQL Servlet home page. "Running the XSQL Demos" on page 14-10 explains how to access the home page.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Demo Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>home/</td>
<td>XSQL Pages &amp; XSQL Servlet</td>
<td>Contains the pages that display the tabbed home page of the XSQL demos as well as the online XSQL help that you can access from that page. As explained in &quot;Running the XSQL Demos&quot; on page 14-10, you can invoke the XSQL home page from the index.html page.</td>
</tr>
<tr>
<td>helloworld/</td>
<td>Hello World Page</td>
<td>Illustrates the simplest possible XSQL page.</td>
</tr>
<tr>
<td>emp/</td>
<td>Employee Page</td>
<td>XSQL page showing XML data from the hr.employees table, using XSQL page parameters to control what employees are returned and which columns to use for the database sort. Uses an associated XSLT Stylesheet to format the results as an HTML Form containing the emp.xsql page as the form action so the user can refine the search criteria.</td>
</tr>
<tr>
<td>insclaim/</td>
<td>Insurance Claim Page</td>
<td>Demonstrates a number of sample queries over the richly-structured Insurance Claim object view. The insclaim.sql scripts set up the INSURANCE_CLAIM_VIEW object view and populates it with sample data.</td>
</tr>
<tr>
<td>classerr/</td>
<td>Invalid Classes Page</td>
<td>Uses invalidclasses.xsl to format a &quot;live&quot; list of current Java class compilation errors in your schema. The accompanying SQL script sets up the XSQLJavaClassesView object view used by the demo. The master/detail information from the object view is formatted into HTML by the invalidclasses.xsl stylesheet in the server.</td>
</tr>
<tr>
<td>doyouxml/</td>
<td>Do You XML? Site</td>
<td>Shows how a simple, data-driven Web site can be built with an XSQL page that makes use of SQL, XSQL substitution variables in the queries, and XSLT for formatting the site. Demonstrates using substitution parameters in both the body of SQL query statements within <a href="">xsql:query</a> tags, as well as within the attributes to <a href="">xsql:query</a> tags to control behavior such as how many records to display and to skip (for &quot;paging&quot; through query results in a stateless way).</td>
</tr>
<tr>
<td>empdept/</td>
<td>Emp/Dept Object Demo</td>
<td>Demonstrates how to use an object view to group master/detail information from two existing flat tables such as scott.emp and scott.dept. The empdeptobjs.sql script creates the object view as well as INSTEAD OF INSERT triggers that enable the master/detail view to be used as an insert target of xsql:insert-request. The empdept.xsl stylesheet illustrates a form of an XSLT stylesheet that looks just like an HTML page without the extra xsl:stylesheet or xsl:transform at the top. Using a Literal Result Element as Stylesheet is part of the XSLT 1.0 specification. The stylesheet also shows how to generate an HTML page that includes &lt;link rel=&quot;stylesheet&quot;&gt; to enable the generated HTML to fully leverage CSS for centralized HTML style information, found in the coolcolors.css file.</td>
</tr>
</tbody>
</table>
Setting Up the XSQL Demos

To set up the XSQL demos perform the following steps:

1. Change into the $ORACLE_HOME/xdk/demo/java/xsql directory (UNIX) or %ORACLE_HOME%\xdk\demo\java\xsql directory (Windows).

### Table 14–1 (Cont.) XSQL Servlet Demos

<table>
<thead>
<tr>
<th>Directory</th>
<th>Demo Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>airport/</td>
<td>Airport Code</td>
<td>Returns a datagram of information about airports based on their three-letter codes and uses (<a href="">xsql:no-rows-query</a>) as alternative queries when initial queries return no rows. After attempting to match the airport code passed in, the XSQL page tries a fuzzy match based on the airport description. The airport.htm page shows how to use the XML results of the airport.xsql page from a Web page with JavaScript to exploit built-in DOM functionality in Internet Explorer. When you enter the three-letter airport code on the Web page, a JavaScript fetches an XML datagram from XSQL servlet. The datagram corresponds to the code that you entered. If the return indicates no match, then the program collects a ‘picklist’ of possible matches based on information returned in the XML datagram from XSQL servlet.</td>
</tr>
<tr>
<td>airport/</td>
<td>Airport Code</td>
<td>Demonstrates use of the same XSQL page as the Airport Code Validation example but supplies an XSLT stylesheet name in the request. This behavior causes the airport information to be formatted as an HTML form instead of being returned as raw XML.</td>
</tr>
<tr>
<td>airport/</td>
<td>Airport Soap</td>
<td>Demonstrates returning airport information as a SOAP Service.</td>
</tr>
<tr>
<td>adhocsql/</td>
<td>Adhoc Query</td>
<td>Demonstrates how to pass a SQL query and XSLT stylesheet as parameters to the server.</td>
</tr>
<tr>
<td>document/</td>
<td>XML Document</td>
<td>Demonstrates inserting XML documents into relational tables. The docdemo.sql script creates a user-defined type called XMLDOCFRAG containing an attribute of type CLOB. Try inserting the text of the document in ./xsql/demo/xml99.xml and providing the name xml99.xsl as the stylesheet, as well as ./xsql/demo/JDevRelNotes.xml with the stylesheet relnotes.xsl. The docstyle.xsql page illustrates an example of the (<a href="">xsql:include-xsql</a>) action element to include the output of the doc.xsql page into its own page before transforming the final output using a client-supplied stylesheet name. The demo uses the client-side XML features of Internet Explorer 5.0 to check the document for well-formedness before allowing it to be posted to the server.</td>
</tr>
<tr>
<td>insertxml/</td>
<td>XML Insert Request</td>
<td>Demonstrates posting XML from a client to an XSQL page that handles inserting the posted XML data into a database table using the (<a href="">xsql:insert-request</a>) action element. The demo accepts XML documents in the moreover.com XML-based news format. In this case, the program doing the posting of the XML is a client-side Web page using Internet Explorer 5.0 and the XMLHttpRequest object from JavaScript. If you look at the source for the insertnewstory.xsql page, you’ll see it’s specifying a table name and an XSLT Transform name. The moreover-to-newsstory.xsl stylesheet transforms the incoming XML information into the canonical format that the OracleXMLSave utility knows how to insert. Try copying and pasting the example (&lt;article&gt;) element several times within the (&lt;moreovernews&gt;) element to insert several new articles in one shot. The newstory.sql script shows how INSTEAD OF triggers can be used on the database views into which you ask XSQL Pages to insert to the data to customize how incoming data is handled, default primary key values, and so on.</td>
</tr>
<tr>
<td>svg/</td>
<td>Scalable Vector</td>
<td>The deptlist.xsql page displays a simple list of departments with hyperlinks to the SalChart.xsql page. The SalChart.xsql page queries employees for a given department passed in as a parameter and uses the associated SalChart.xsql stylesheet to format the result into a Scalable Vector Graphics drawing, a bar chart comparing salaries of the employees in that department.</td>
</tr>
<tr>
<td>fop/</td>
<td>PDF</td>
<td>The emptable.xsql page displays a simple list of employees. The emptable.xsl stylesheet transforms the datapage into the XSL-FO Formatting Objects which, combined with the built-in FOP serializer, render the results in Adobe PDF format.</td>
</tr>
<tr>
<td>cursor/</td>
<td>Cursor</td>
<td>Contains an example of using a nested (<a href="">xsql:query</a>) expression, which is one of three ways to use the default (<a href="">xsql:query</a>) element to produce nested elements.</td>
</tr>
<tr>
<td>actions/</td>
<td></td>
<td>Contains the source code for two example custom actions.</td>
</tr>
</tbody>
</table>
2. Start SQL*Plus and connect to your database as ctxsys — the schema owner for the Oracle Text packages — and issue the following statement:

   GRANT EXECUTE ON ctx_ddl TO scott;

3. Connect to your database as a user with DBA privileges and issue the following statement:

   GRANT QUERY REWRITE TO scott;

   The preceding query enables scott to create a function-based index that one of the demos requires to perform case-insensitive queries on descriptions of airports.

4. Connect to your database as scott. You will be prompted for the password.

5. Run the SQL script install.sql in the current directory. This script runs all SQL scripts for all the demos:

   @install.sql

6. Change to the ./doyouxml subdirectory, and run the following command to import sample data for the "Do You XML?" demo (you will be prompted for the password):

   imp scott file=doyouxml.dmp

7. To run the Scalable Vector Graphics (SVG) demonstration, install an SVG plug-in such as Adobe SVG plug-in into your browser.

**Running the XSQL Demos**

The XSQL demos are designed to be accessed through a Web browser. If you have set up the XSQL servlet in a Web server as described in "Configuring the XSQL Servlet Container" on page 14-6, then you can access the demos through the following URL, substituting appropriate values for yourserver and port:

   http://yourserver:port/xsql/index.html

   Figure 14–3 shows a section of the XSQL home page in Internet Explorer. Note that you must use browser version 5 or higher.
Using the XSQL Pages Publishing Framework: Overview

Figure 14–3 XSQL Home Page

The demos are designed to be self-explanatory. Click the demo titles—Hello World Page, Employee Page, and so forth—and follow the online instructions.

Using the XSQL Pages Command-Line Utility

Often the content of a dynamic page is based on data that does not frequently change. To optimize performance of your Web publishing, you can use operating system facilities to schedule offline processing of your XSQL pages. This technique enables the processed results to be served statically by your Web server.

The XDK includes a command-line Java interface that runs the XSQL page processor. You can process any XSQL page with the XSQL command-line utility.

The $ORACLE_HOME/xdk/bin/xsql and %ORACLE_HOME%/xdk/bin/xsql.bat shell scripts run the oracle.xml.xsql.XSQLCommandLine class. Before invoking the class make sure that your environment is configured as described in "Setting Up the XSQL Pages Framework" on page 14-5. Depending on how you invoke the utility, the syntax is either of the following:

```
java oracle.xml.xsql.XSQLCommandLine xsqlpage [outfile] [param1=value1 ...]
xsql xsqlpage [outfile] [param1=value1 ...]
```
If you specify an `outfile`, then the result of processing `xsqlpage` is written to it; otherwise the result goes to standard out. You can pass any number of parameters to the XSQL page processor, which are available for reference by the XSQL page processed as part of the request. However, the following parameter names are recognized by the command-line utility and have a pre-defined behavior:

- `xml-stylesheet=stylesheetURL`
  Provides the relative or absolute URL for a stylesheet to use for the request. You can also set it to the string `none` to suppress XSLT stylesheet processing for debugging purposes.

- `posted-xml=XMLDocumentURL`
  Provides the relative or absolute URL of an XML resource to treat as if it were posted as part of the request.

- `useragent=UserAgentString`
  Simulates a particular HTTP User-Agent string from the command line so that an appropriate stylesheet for that User-Agent type is selected as part of command-line processing of the page.

Generating and Transforming XML with XSQL Servlet

This section describes the most basic tasks that you can perform with your server-side XSQL page templates:

- Composing XSQL Pages
- Producing Datagrams from SQL Queries
- Transforming XML Datagrams into an Alternative XML Format
- Transforming XML Datagrams into HTML for Display

Composing XSQL Pages

You can serve database information in XML format over the Web with XSQL pages. For example, suppose your aim is to serve a real-time XML datagram from Oracle of all available flights landing today at JFK airport. Example 14–3 shows a sample XSQL page in a file named `AvailableFlightsToday.xsql`.

Example 14–3  AvailableFlightsToday.xsql

```xml
<?xml version="1.0"?>
<xsql:query connection="demo" bind-params="City" xmlns:xsql="urn:oracle-xsql">
  SELECT    Carrier, FlightNumber, Origin, TO_CHAR(ExpectedTime,'HH24:MI') AS Due
  FROM      FlightSchedule
  WHERE     TRUNC(ExpectedTime) = TRUNC(SYSDATE)
  AND       Arrived = 'N'
  AND       Destination = ?   /* The '?' represents a bind variable bound */
  ORDER BY  ExpectedTime      /* to the value of the City parameter. */
</xsql:query>
```

The XSQL page is an XML file that contains any mix of static XML content and XSQL action elements. The file can have any extension, but `.xsql` is the default extension for XSQL pages. You can modify your servlet engine configuration settings to associate other extensions by using the same technique described in "Configuring the XSQL Servlet Container" on page 14-6. Note that the servlet extension mapping is configured inside the `.WEB-INF/web.xml` file in a J2EE WAR file.
The XSQL page in Example 14–3 begins with the following declaration:

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

The first, outermost element in an XSQL page is the **document element**. AvailableFlightsToday.xsql contains a single XSQL action element `<xsql:query>`, but no static XML elements. In this case the `<xsql:query>` element is the document element. Example 14–3 represents the simplest useful XSQL page: one that contains a single query. The results of the query replace the `<xsql:query>` section in the XSQL page.

---

**Note:** Chapter 30, "XSQL Pages Reference" describes the complete set of built-in action elements.

---

The `<xsql:query>` action element includes an `xmlns` attribute that declares the `xsql` namespace prefix as a synonym for the `urn:oracle-xsql` value, which is the Oracle XSQL namespace identifier:

```xml
<xsql:query connection="demo" bind-params="City" xmlns:xsql="urn:oracle-xsql">
```

The element also contains a `connection` attribute whose value is the name of one of the pre-defined connections in the XSQL configuration file:

```xml
<xsql:query connection="demo" bind-params="City" xmlns:xsql="urn:oracle-xsql">
```

The details concerning the username, password, database, and JDBC driver that will be used for the `demo` connection are centralized in the configuration file.

To include more than one query on the page, you can invent an XML element to wrap the other elements. Example 14–4 illustrates this technique.

**Example 14–4  Wrapping the `<xsql:query>` Element**

```xml
<?xml version="1.0"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql">

<xsql:query bind-params="City">

    SELECT   Carrier, FlightNumber, Origin, TO_CHAR(ExpectedTime,'HH24:MI') AS Due
    FROM     FlightSchedule
    WHERE    TRUNC(ExpectedTime) = TRUNC(SYSDATE)
    AND      Arrived = 'N'
    AND      Destination = ? /* The ? is a bind variable bound to the value of the City parameter. */
    ORDER BY ExpectedTime /* to the value of the City parameter. */

</xsql:query>

<!-- Other xsql:query actions can go here inside <page> and </page> -->
</page>
```

In Example 14–4, the `connection` attribute and the `xsql` namespace declaration always go on the document element, whereas the `bind-params` is specific to the `<xsql:query>` action.

**Using Bind Parameters**

The `<xsql:query>` element shown in Example 14–3 contains a `bind-params` attribute that associates the values of parameters in the request to bind variables in the SQL statement included in the `<xsql:query>` tag. The bind parameters in the SQL statement are represented by question marks.

You can use SQL bind variables to parameterize the results of any of the actions in Table 30–1, "Built-In XSQL Elements and Action Handler Classes" that allow SQL
statements. Bind variables enable your XSQL page template to produce results based on the values of parameters passed in the request.

To use a bind variable, include a question mark anywhere in a statement where bind variables are allowed by SQL. Whenever a SQL statement is executed in the page, the XSQL engine binds the parameter values to the variable by specifying the bind-params attribute on the action element.

Example 14–5 illustrates an XSQL page that binds the bind variables to the value of the custid parameter in the page request.

Example 14–5 CustomerPortfolio.xsql

```xml
<portfolio connection="prod" xmlns:xsql="urn:oracle-xsql">
  <xsql:query bind-params="custid">
    SELECT s.ticker as 'Symbol', s.last_traded_price as 'Price'
    FROM latest_stocks s, customer_portfolio p
    WHERE p.customer_id = ?
    AND s.ticker = p.ticker
  </xsql:query>
</portfolio>
```

The XML data for a customer with ID of 101 can then be requested by passing the customer id parameter in the request as follows:


The value of the bind-params attribute is a space-delimited list of parameter names. The left-to-right order indicates the positional bind variable to which its value will be bound in the statement. Thus, if your SQL statement contains five question marks, then the bind-params attribute needs a space-delimited list of five parameter names. If the same parameter value needs to be bound to several different occurrences of a bind variable, then repeat the name of the parameters in the value of the bind-params attribute at the appropriate position. Failure to include the same number of parameter names in the bind-params attribute as in the query results in an error when the page is executed.

You can use variables in any action that expects a SQL statement or PL/SQL block. The page shown in Example 14–6 illustrates this technique. The XSQL page contains three action elements:

- `<xsql:dml>` binds useridCookie to an argument in the log_user_hit procedure.
- `<xsql:query>` binds parameter custid to a variable in a WHERE clause.
- `<xsql:include-owa>` binds parameters custid and userCookie to two arguments in the historical_data procedure.

Example 14–6 CustomerPortfolio.xsql

```xml
<portfolio connection="prod" xmlns:xsql="urn:oracle-xsql">
  <xsql:dml commit='yes' bind-params="useridCookie">
    BEGIN log_user_hit(?); END;
  </xsql:dml>
  <current-prices>
    <xsql:query bind-params="custid">
      SELECT s.ticker as 'Symbol', s.last_traded_price as 'Price'
      FROM latest_stocks s, customer_portfolio p
      WHERE p.customer_id = ?
      AND s.ticker = p.ticker
    </xsql:query>
  </current-prices>
</portfolio>
```
Using Lexical Substitution Parameters

For any XSQL action element, you can substitute the value of any attribute or the text of any contained SQL statement by means of a lexical substitution parameter. Thus, you can parameterize how actions behave as well as substitute parts of the SQL statements that they perform. Lexical substitution parameters are referenced with the following syntax: {@ParameterName}.

Example 14–7 illustrates how you can use two lexical substitution parameters. One parameter in the <xsql:query> element sets the maximum number of rows to be passed in, whereas the other controls the list of columns to be ordered.

Example 14–7 DevOpenBugs.xsql

<!-- DevOpenBugs.xsql -->
<open-bugs connection="demo" xmlns:xsql="urn:oracle-xsql">
  <xsql:query max-rows="{@max}" bind-params="dev prod">
    SELECT bugno, abstract, status
    FROM bug_table
    WHERE programmer_assigned = UPPER(?)
    AND product_id = ?
    AND status < 80
    ORDER BY {orderby}
  </xsql:query>
</open-bugs>

Example 14–7 also contains two bind parameters: dev and prod. Suppose that you want to obtain the open bugs for developer smuench against product 817. You want to retrieve only 10 rows and order them by bug number. You can fetch the XML for the bug list by specifying parameter values as follows:


You can also use the XSQL command-line utility to make the request as follows:

xsq1 DevOpenBugs.xsql dev=smuench prod=817 max=10 orderby=bugno

Lexical parameters also enable you to parameterize the XSQL pages connection and the stylesheet used to process the page. Example 14–8 illustrates this technique. You can switch between stylesheets test.xsql and prod.xsl by specifying the name/value pairs sheet=test and sheet=prod.

Example 14–8 DevOpenBugs.xsql

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="{@sheet}.xsl"?>
<!-- DevOpenBugs.xsql -->
<open-bugs connection="{@conn}" xmlns:xsql="urn:oracle-xsql">
  <xsql:query max-rows="{@max}" bind-params="dev prod">
    SELECT bugno, abstract, status
    FROM bug_table
    WHERE programmer_assigned = UPPER(?)
    AND product_id = ?
  </xsql:query>
</open-bugs>
Providing Default Values for Bind and Substitution Parameters

You may want to provide a default value for a bind variable or a substitution parameter directly in the page. In this way, the page is parameterized without requiring the requester to explicitly pass in all values in every request.

To include a default value for a parameter, add an XML attribute of the same name as the parameter to the action element or to any ancestor element. If a value for a given parameter is not included in the request, then the XSQL page processor searches for an attribute by the same name on the current action element. If it does not find one, it keeps looking for such an attribute on each ancestor element of the current action element until it gets to the document element of the page.

The page in Example 14–9 defaults the value of the max parameter to 10 for both <xsql:query> actions in the page.

Example 14–9 Setting a Default Value

```xml
<example max="10" connection='demo' xmlns:xsql='urn:oracle-xsql'>
  <xsql:query max-rows="{@max}">SELECT * FROM TABLE1</xsql:query>
  <xsql:query max-rows="{@max}">SELECT * FROM TABLE2</xsql:query>
</example>
```

This page in Example 14–10 defaults the first query to a max of 5, the second query to a max of 7, and the third query to a max of 10.

Example 14–10 Setting Multiple Default Values

```xml
<example max="10" connection='demo' xmlns:xsql='urn:oracle-xsql'>
  <xsql:query max="5" max-rows="{@max}">SELECT * FROM TABLE1</xsql:query>
  <xsql:query max="7" max-rows="{@max}">SELECT * FROM TABLE2</xsql:query>
  <xsql:query max-rows="{@max}">SELECT * FROM TABLE3</xsql:query>
</example>
```

All defaults are overridden if a value of max is supplied in the request, as shown in the following example:

```
http://yourserver.com/example.xsql?max=3
```

Bind variables respect the same defaulting rules. Example 14–11 illustrates how you can set the val parameter to 10 by default.

Example 14–11 Defaults for Bind Variables

```xml
<example val="10" connection='demo' xmlns:xsql='urn:oracle-xsql'>
  <xsql:query tag-case="lower" bind-params="val val val">
    SELECT ? AS somevalue
    FROM DUAL
    WHERE ? = ?
  </xsql:query>
</example>
```

If the page in Example 14–11 is requested without any parameters, it returns the following XML datagram:
Alternatively, assume that the page is requested with the following URL:

```
http://yourserver.com/example.xsql?val=3
```

The preceding URL returns the following datagram:

```
<example>
  <rowset>
    <row>
      <somevalue>3</somevalue>
    </row>
  </rowset>
</example>
```

You can remove the default value for the `val` parameter from the page by removing the `val` attribute. Example 14–12 illustrates this technique.

**Example 14–12  Bind Variables with No Defaults**

```
<example connection="demo" xmlns:xsql="urn:oracle-xsql">
  <xsql:query tag-case="lower" bind-params="val val val">
    SELECT ? AS somevalue 
    FROM DUAL
    WHERE ? = ?
  </xsql:query>
</example>
```

A URL request for the page that does not supply a name/value pair returns the following datagram:

```
<example>
  <rowset/>
</example>
```

A bind variable that is bound to a parameter with *neither* a default value *nor* a value supplied in the request is bound to NULL, which causes the `WHERE` clause in Example 14–12 to return no rows.

**How the XSQL Page Processor Handles Different Types of Parameters**

XSQL pages can make use of parameters supplied in the request as well as page-private parameters. The names and values of page-private parameters are determined by actions in the page. If an action encounters a reference to a parameter named `param` in either a `bind-params` attribute or in a lexical parameter reference, then the value of the `param` parameter is resolved in the following order:

1. The value of the page-private parameter named `param`, if set
2. The value of the request parameter named `param`, if supplied
3. The default value provided by an attribute named `param` on the current action element or one of its ancestor elements
4. The value NULL for bind variables and the empty string for lexical parameters
For XSQL pages that are processed by the XSQL servlet over HTTP, you can also set and reference the HTTP-Session-level variables and HTTP Cookies parameters.

For XSQL pages processed through the XSQL servlet, the value of a parameter param is resolved in the following order:

1. The value of the page-private parameter param, if set
2. The value of the cookie named param, if set
3. The value of the session variable named param, if set
4. The value of the request parameter named param, if supplied
5. The default value provided by an attribute named param on the current action element or one of its ancestor elements
6. The value NULL for bind variables and the empty string for lexical parameters

The resolution order means that users cannot supply parameter values in a request to override parameters of the same name set in the HTTP session. Also, users cannot set them as cookies that persist across browser sessions.

**Producing Datagrams from SQL Queries**

With XSQL servlet properly installed on your Web server, you can access XSQL pages by following these basic steps:

1. Copy an XSQL file to a directory under the virtual hierarchy of your Web server. Example 14–3 shows the sample page AvailableFlightsToday.xsql.

   You can also deploy XSQL pages in a standard J2EE WAR file, which occurs when you use Oracle JDeveloper 10g to develop and deploy your pages to Oracle Application Server.

2. Load the page in your browser. For example, if the root URL is yourcompany.com, then you can access the AvailableFlightsToday.xsql page through a Web browser by requesting the following URL:

   http://yourcompany.com/AvailableFlightsToday.xsql?City=JFK

   The XSQL page processor automatically materializes the results of the query in your XSQL page as XML and returns them to the requester. Typically, another server program requests this XML-based datagram for processing, but if you use a browser such as Internet Explorer, then you can directly view the XML result as shown in Figure 14–4.
Transforming XML Datagrams into an Alternative XML Format

If the canonical <ROWSET> and <ROW> XML output from Figure 14–4 is not the XML format you need, then you can associate an XSLT stylesheet with your XSQL page. The stylesheet can transform the XML datagram in the server before returning the data.

When exchanging data with another program, you typically agree on a DTD that describes the XML format for the exchange. Assume that you are given the flight-list.dtd definition and are told to produce your list of arriving flights in a format compliant with the DTD. You can use a visual tool such as XML Authority to browse the structure of the flight-list DTD, as shown in Figure 14–5.
Figure 14–5 Exploring flight-list.dtd with XML Authority

Figure 14–5 shows that the standard XML formats for flight lists are as follows:

■ <flight-list> element, which contains one or more <flight> elements
■ <flight> elements, which have attributes airline and number, and each of which contains an <arrives> element
■ <arrives> elements, which contains text

Example 14–13 shows the XSLT stylesheet flight-list.xsl. By associating the stylesheet with the XSQL page, you can change the default <ROWSET> and <ROW> format into the industry-standard <flight-list> and <flight>.

Example 14–13 flight-list.xsl

<!-- XSLT Stylesheet to transform ROWSET/ROW results into flight-list format
-->  
<flight-list xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xsl:version="1.0">
  <xsl:for-each select="ROWSET/ROW">
    <flight airline="{CARRIER}" number="{FLIGHTNUMBER}"
        >
      <arrives><xsl:value-of select="DUE"/></arrives>
    </flight>
  </xsl:for-each>
</flight-list>

The XSLT stylesheet is a template that includes the literal elements that you want to produce in the resulting document, such as <flight-list>, <flight>, and <arrives>, interspersed with XSLT actions that enable you to do the following:

■ Loop over matching elements in the source document with <xsl:for-each>
■ Plug in the values of source document elements where necessary with <xsl:value-of>
■ Plug in the values of source document elements into attribute values with the {some_parameter} notation

The following items have been added to the top-level <flight-list> element in the Example 14–13 stylesheet:
This attribute defines the XML Namespace named xsl and identifies the URL string that uniquely identifies the XSLT specification. Although it looks just like a URL, think of the string http://www.w3.org/1999/XSL/Transform as the “global primary key” for the set of elements defined in the XSLT 1.0 specification. When the namespace is defined, you can use the <xsl:XXX> action elements in the stylesheet to loop and plug values in where necessary.

This attribute identifies the document as an XSLT 1.0 stylesheet. A version attribute is required on all XSLT stylesheets for them to be valid and recognized by an XSLT processor.

You can associate the flight-list.xsl stylesheet with the AvailableFlightsToday.xsql in Example 14–3 by adding an <?xml-stylesheet?> instruction to the top of the page. Example 14–14 illustrates this technique.

Example 14–14 flight-list.xsl

```xml
<?xml version="1.0"?>
<xml-stylesheet type=\"text/xsl\" href=\"flight-list.xsl\">
<sql:query connection=\"demo\" bind-params=\"City\" xmlns:xsql=\"urn:oracle-xsql\">
  SELECT Carrier, FlightNumber, Origin, TO_CHAR(ExpectedTime,\'HH24:MI\') AS Due
  FROM FlightSchedule
  WHERE TRUNC(ExpectedTime) = TRUNC(SYSDATE) AND Arrived = \"N\"
  AND Destination = ? /* The ? is a bind variable being bound */
  ORDER BY ExpectedTime /* to the value of the City parameter */
</sql:query>
```

Associating an XSLT stylesheet with the XSQL page causes the requesting program or browser to view the XML in the format as specified by flight-list.dtd you were given. Figure 14–6 illustrates a sample browser display.

Figure 14–6 XSQL Page Results in XML Format

```
<?xml version=\"1.0\" encoding=\"UTF-8\" ?>
- <flight-list>
  - <flight airline=\"VS\" number=\"344\">
    <arrives>16:10</arrives>
  </flight>
  - <flight airline=\"LH\" number=\"466\">
    <arrives>21:33</arrives>
  </flight>
  - <flight airline=\"UA\" number=\"32\">
    <arrives>23:54</arrives>
  </flight>
</flight-list>
```
Transforming XML Datagrams into HTML for Display

To return the same XML data in HTML instead of an alternative XML format, use a different XSLT stylesheet. For example, rather than producing elements such as `<flight-list>` and `<flight>`, you can write a stylesheet that produces HTML elements such as `<table>`, `<tr>`, and `<td>`. The result of the dynamically queried data then looks like the HTML page shown in Figure 14–7. Instead of returning raw XML data, the XSQL page leverages server-side XSLT transformation to format the information as HTML for delivery to the browser.

Figure 14–7 Using an XSLT Stylesheet to Render HTML

Similar to the syntax of the `flight-list.xsl` stylesheet, the `flight-display.xsl` stylesheet shown in Example 14–15 looks like a template HTML page. It contains `<xsl:for-each>` and `<xsl:value-of>`, and attribute value templates such as `{DUE}` to plug in the dynamic values from the underlying `<ROWSET>` and `<ROW>` structured XML query results.

Example 14–15  `flight-display.xsl`

```xml
<!-- XSLT Stylesheet to transform ROWSET/ROW results into HTML -->
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xsl:version="1.0">
<head><link rel="stylesheet" type="text/css" href="flights.css" /></head>
<body>
  <center>
    <table border="0">
      <tr><th>Flight</th><th>Arrives</th></tr>
      <xsl:for-each select="ROWSET/ROW">
        <tr>
          <td>
            <table border="0" cellspacing="0" cellpadding="4">
              <tr><td><img align="absmiddle" src="images/{CARRIER}.gif" /></td>
              <td width="180">
                <xsl:value-of select="CARRIER" />
              </td></tr>
            </table>
          </td>
        </tr>
      </xsl:for-each>
    </table>
  </center>
</body>
</html>
```
You can achieve useful results quickly by combining the power of the following:

- Parameterized SQL statements to select information from the Oracle database
- Industry-standard XML as a portable, interim data exchange format
- XSLT to transform XML-based datagrams into any XML- or HTML-based format

### Using XSQL in Java Programs

The `oracle.xml.xsql.XSQLRequest` class enables you to use the XSQL page processor in your Java programs. To use the XSQL Java API, follow these basic steps:

1. Construct an instance of `XSQLRequest`, passing the XSQL page to be processed into the constructor as one of the following:
   - String containing a URL to the page
   - URL object for the page
   - In-memory `XMLDocument`

2. Invoke one of the following methods on the object to process the page:
   - `process()` to write the result to a `PrintWriter` or `OutputStream`
   - `processToXML()` to return the result as an XML Document

If you want to use the built-in XSQL connection manager, which implements JDBC connection pooling based on XSQL configuration file definitions, then the XSQL page is all you need to pass to the constructor. Optionally, you can pass in a custom implementation for the `XSQLConnectionManagerFactory` interface as well.

The ability to pass the XSQL page as an in-memory `XMLDocument` object means that you can dynamically generate any valid XSQL page for processing. You can then pass the page to the XSQL engine for evaluation.

When processing a page, you may want to perform the following additional tasks as part of the request:

- Pass a set of parameters to the request.

  You accomplish this aim by passing any object that implements the `Dictionary` interface to the `process()` or `processToXML()` methods. Passing a `HashTable` containing the parameters is one popular approach.
Set an XML document to be processed by the page as if it were the "posted XML" message body.

You can do this by using the `XSQLRequest.setPostedDocument()` method.

Example 14–16 shows how you can process a page by using `XSQLRequest`.

Example 14–16  XSQLRequestSample Class

```java
import oracle.xml.xsql.XSQLRequest;
import java.util.Hashtable;
import java.io.PrintWriter;
import java.net.URL;
public class XSQLRequestSample {
    public static void main(String[] args) throws Exception {
        // Construct the URL of the XSQL Page
        URL pageUrl = new URL("file:///C:/foo/bar.xsql");
        // Construct a new XSQL Page request
        XSQLRequest req = new XSQLRequest(pageUrl);
        // Set up a Hashtable of named parameters to pass to the request
        Hashtable params = new Hashtable(3);
        params.put("param1", "value1");
        params.put("param2", "value2");
        /* If needed, treat an existing, in-memory XMLDocument as if
         * it were posted to the XSQL Page as part of the request
         * req.setPostedDocument(myXMLDocument);
         * 
         */
        // Process the page, passing the parameters and writing the output
        // to standard out.
        req.process(params, new PrintWriter(System.out),
                    new PrintWriter(System.err));
    }
}
```

See Also: Chapter 15, "Using the XSQL Pages Publishing Framework: Advanced Topics" to learn more about the XSQL Java API

XSQL Pages Tips and Techniques

This section contains the following topics:

- XSQL Pages Limitations
- Hints for Using the XSQL Servlet
- Resolving Common XSQL Connection Errors
- Security Considerations for XSQL Pages

XSQL Pages Limitations

HTTP parameters with multibyte names, for example, a parameter whose name is in Kanji, are properly handled when they are inserted into your XSQL page with the `<xsql:include-request-params>` element. An attempt to refer to a parameter with a multibyte name inside the query statement of an `<xsql:query>` tag returns an empty string for the parameter value.

As a workaround use a nonmultibyte parameter name. The parameter can still have a multibyte value that can be handled correctly.
Hints for Using the XSQL Servlet

This section lists the following XSQL Servlet hints:

- Specifying a DTD While Transforming XSQL Output to a WML Document
- Testing Conditions in XSQL Pages
- Passing a Query Result to the WHERE Clause of Another Query
- Handling Multi-Valued HTML Form Parameters
- Invoking PL/SQL Wrapper Procedures to Generate XML Datagrams
- Accessing Contents of Posted XML
- Changing Database Connections Dynamically
- Retrieving the Name of the Current XSQL Page

Specifying a DTD While Transforming XSQL Output to a WML Document

You can specify a DTD while transforming XSQL output to a WML document for a wireless application. The technique is to use a built-in facility of the XSLT stylesheet called `<xsl:output>`. The following example illustrates this technique:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output type="xml" doctype-system="your.dtd"/>
  <xsl:template match="/">
  ...  
  </xsl:template>
</xsl:stylesheet>
```

The preceding stylesheet produces an XML result that includes the following code, where "your.dtd" can be any valid absolute or relative URL:

```
<!DOCTYPE xxxx SYSTEM "your.dtd">
```

Testing Conditions in XSQL Pages

You can include if-then logic in your XSQL pages. Example 14–17 illustrates a technique for executing a query based on a test of a parameter value.

```
<xsql:if-param name="security" equals="admin">
  <xsql:query>
    SELECT ....
  </xsql:query>
</xsql:when>
<xsql:if-param name="security" equals="user">
  <xsql:query>
    SELECT ....
  </xsql:query>
</xsql:if-param>
```

Passing a Query Result to the WHERE Clause of Another Query

If you have two queries in an XSQL page, then you can use the value of a select list item of the first query in the second query by using page parameters. Example 14–18 illustrates this technique.
Example 14–18  Passing Values Among SQL Queries

```xml
<page xmlns:xsql="urn:oracle-xsql" connection="demo">
  <!-- Value of page param 'xxx' will be first column of first row -->
  <xsql:set-page-param name="xxx">
    SELECT one FROM table1 WHERE ...
  </xsql:set-param-param>
  <xsql:query bind-params="xxx">
    SELECT col3,col4 FROM table2
    WHERE col3 = ?
  </xsql:query>
</page>
```

Handling Multi-Valued HTML Form Parameters

In some situations you may need to process multi-valued HTML `<form>` parameters that are needed for `<input name="choices" type="checkbox">`. Use the parameter array notation on your parameter name (for example, `choices[]`) to refer to the array of values from the selected check boxes.

Assume that you have a multi-valued parameter named `guy`. You can use the array parameter notation in an XSQL page as shown in Example 14–19.

Example 14–19  Handling Multi-Valued Parameters

```xml
<page xmlns:xsql="urn:oracle-xsql">
  <xsql:set-page-param name="guy-list" value="{guy[]}" treat-list-as-array="yes"/>
  <xsql:set-page-param name="quoted-guys" value="{guy[]}" quote-array-values="yes" treat-list-as-array="yes"/>
  <xsql:include-param name="guy-list"/>
  <xsql:include-param name="quoted-guys"/>
  <xsql:include-param name="guy[]"/>
</page>
```

Assume that you request this page is requested with the following URL, which contains multiple parameters of the same name to produce a multi-valued attribute:

```
http://yourserver.com/page.xsql?guy=Curly&guy=Larry&guy=Moe
```

The page returned looks like the following:

```
<page>
  <guy-list>Curly,Larry,Moe</guy-list>
  <quoted-guys>'Curly','Larry','Moe'</quoted-guys>
  <guy>
    <value>Curly</value>
    <value>Larry</value>
    <value>Moe</value>
  </guy>
</page>
```

You can also use the value of a multi-valued page parameter in a SQL statement `WHERE` clause by using the code shown in Example 14–20.

Example 14–20  Using Multi-Valued Page Parameters in a SQL Statement

```xml
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
  <xsql:set-page-param name="quoted-guys" value="{guy[]}"
    treat-list-as-array="yes"
    quote-array-values="yes"/>
  <xsql:query>
```

14-26  Oracle XML Developer's Kit Programmer's Guide
SELECT *
FROM sometable
WHERE name IN (@quoted-guys)
</xsql:query>
</page>

Invoking PL/SQL Wrapper Procedures to Generate XML Datagrams

You cannot set parameter values by binding them in the position of OUT variables with <xsql:dml>. Only IN parameters are supported for binding. You can create a wrapper procedure, however, that constructs XML elements with the HTTP package. Your XSQL page can then invoke the wrapper procedure with <xsql:include-owa>.

Example 14–21 shows a PL/SQL procedure that accepts two IN parameters, multiplies them and puts the value in one OUT parameter, then adds them and puts the result in a second OUT parameter.

Example 14–21 addmult PL/SQL Procedure

```sql
CREATE OR REPLACE PROCEDURE addmult(arg1 NUMBER, arg2 NUMBER, sumval OUT NUMBER, prodval OUT NUMBER)
IS
BEGIN
  sumval := arg1 + arg2;
  prodval := arg1 * arg2;
END;
```

You can write the PL/SQL procedure in Example 14–22 to wrap the procedure in Example 14–21. The addmultwrapper procedure accepts the IN arguments that the addmult procedure preceding expects, and then encodes the OUT values as an XML datagram that you print to the OWA page buffer.

Example 14–22 addmultwrapper PL/SQL Procedure

```sql
CREATE OR REPLACE PROCEDURE addmultwrapper(arg1 NUMBER, arg2 NUMBER)
IS
  sumval NUMBER;
  prodval NUMBER;
  xml VARCHAR2(2000);
BEGIN
  -- Call the procedure with OUT values
  addmult(arg1, arg2, sumval, prodval);
  -- Then produce XML that encodes the OUT values
  xml := '<addmult>'||
    '<sum>'||sumval||'</sum>'||
    '<product>'||prodval||'</product>'||
  '</addmult>';
  -- Print the XML result to the OWA page buffer for return
  HTP.P(xml);
END;
```

The XSQL page shown in Example 14–23 constructs an XML document by including a call to the PL/SQL wrapper procedure.

Example 14–23 addmult.xsql

```xml
<xsql:include-owa bind-params="arg1 arg2">
  BEGIN addmultwrapper(?,?) END;
</xsql:include-owa>
```
Suppose that you invoke addmult.xsql by entering a URL in a browser as follows:
http://yourserver.com/addmult.xsql?arg1=30&arg2=45

The XML datagram returned by the servlet reflects the `OUT` values as follows:

```
<page>
  <addmult><sum>75</sum><product>1350</product></addmult>
</page>
```

### Accessing Contents of Posted XML
The XSQL page processor can access the contents of posted XML. Any XML document can be posted and handled by the feature that XSQL supports.

For example, an XSQL page can access the contents of an inbound SOAP message by using the `xpath="XpathExpression"` attribute in the `<xsql:set-page-param>` action. Alternatively, custom action handlers can gain direct access to the SOAP message body by calling `getPageRequest().getPostedDocument()`.

To create the SOAP response body to return to the client, use an XSLT stylesheet or a custom serializer implementation to write the XML response in an appropriate SOAP-encoded format.

*See Also:* The Airport SOAP demo for an example of using an XSQL page to implement a SOAP-based Web Service

### Changing Database Connections Dynamically
Suppose that you want to choose database connections dynamically when invoking an XSQL page. For example, you may want to switch between a test database and a production database. You can achieve this goal by including an XSQL parameter in the `connection` attribute of the XSQL page. Make sure to define an attribute of the same name to serve as the default value for the connection name.

Assume that in your XSQL configuration file you define connections for database `testdb` and `proddb`. You then write an XSQL page with the following `<xsql:query>` element:

```
<xsql:query conn='testdb' connection='(@conn) xmlns:xsql="urn:oracle-xsql">
  ...
</xsql:query>
```

If you request this page without any parameters, then the value of the `conn` parameter is `testdb`, so the page uses the connection named `testdb` defined in the XSQL configuration file. If you request the page with `conn=proddb`, then the page uses the connection named `proddb` instead.

### Retrieving the Name of the Current XSQL Page
An XSQL page can access its own name in a generic way at runtime in order to construct links to the current page. You can use a helper method like the one shown in Example 14–24 to retrieve the name of the page inside a custom action handler.

*Example 14–24  Obtaining the Name of the Current XSQL Page*

```java
private String curPageName(XSQLPageRequest req) {
  String thisPage = req.getSourceDocumentURI();
  int pos = thisPage.lastIndexOf('/');
  if (pos >= 0) thisPage = thisPage.substring(pos+1);
  pos = thisPage.indexOf('?');
  return thisPage.substring(0,pos);
}
```
if (pos >=0) thisPage = thisPage.substring(0,pos-1);
return thisPage;
}

Resolving Common XSQL Connection Errors

This section contains tips for responding to XSQL errors:

- Receiving "Unable to Connect" Errors
- Receiving "No Posted Document to Process" When Using HTTP POST

Receiving "Unable to Connect" Errors

Suppose that you are unable to connect to a database and errors similar to the following when running the helloworld.xsql sample program:

Oracle XSQL Servlet Page Processor
XSQL-007: Cannot acquire a database connection to process page.
Connection refused (DESCRIPTION=(TMP=) (VSNNUM=135286784) (ERR=12505) (ERROR_STACK=(ERROR=(CODE=12505) (EMFI=4))))

The preceding errors indicate that the XSQL servlet is attempting the JDBC connection based on the <connectiondef> information for the connection named demo, assuming you did not modify the helloworld.xsql demo page.

By default the XSQLConfig.xml file comes with the entry for the demo connection that looks like the following (use the correct password):

<connection name="demo">
  <username>scott</username>
  <password>password</password>
  <dburl>jdbc:oracle:thin:@localhost:1521:ORCL</dburl>
  <driver>oracle.jdbc.driver.OracleDriver</driver>
</connection>

The error is probably due to one of the following reasons:

- Your database is not on the localhost machine.
- Your database SID is not ORCL.
- Your TNS Listener Port is not 1521.

Receiving "No Posted Document to Process" When Using HTTP POST

When trying to post XML information to an XSQL page for processing, it must be sent by the HTTP POST method. This transfer can be effected by an HTML form or an XML document sent by HTTP POST. If you try to use HTTP GET instead, then there is no posted document, and hence you get the "No posted document to process" error. Use HTTP POST instead to cause the correct behavior.

Security Considerations for XSQL Pages

This section describes best practices for managing security in the XSQL servlet:

- Installing Your XSQL Configuration File in a Safe Directory
- Disabling Default Client Stylesheet Overrides
- Protecting Against the Misuse of Substitution Parameters
Installing Your XSQL Configuration File in a Safe Directory

The XSQLConfig.xml configuration file contains sensitive database username and password information. This file should not reside in any directory that maps to a virtual path of your Web server, nor in any of its subdirectories. The only required permissions for the configuration file are read permission granted to the UNIX account that owns the servlet engine. Failure to follow this recommendation could mean that a user of your site could browse the contents of your configuration file, thereby obtaining the passwords to database accounts.

Disabling Default Client Stylesheet Overrides

By default, the XSQL page processor enables the user to supply a stylesheet in the page request by passing a value for the special xml-stylesheet parameter. If you want the stylesheet referenced by the server-side XSQL page to be the only legal stylesheet, then include the allow-client-style="no" attribute on the document element of your page. You can also globally change the default setting in the XSQLConfig.xml file to disallow client stylesheet overrides. If you take either approach, then the only pages that allow client stylesheet overrides are those that include the allow-client-style="yes" attribute on their document element.

Protecting Against the Misuse of Substitution Parameters

Any product that supports the use of lexical substitution variables in a SQL query can cause a developer problems. Any time you deploy an XSQL page that allows part of all of a SQL statement to be substituted by a lexical parameter, you must make sure that you have taken appropriate precautions against misuse.

For example, one of the demonstrations that comes with XSQL Pages is the Adhoc Query Demo. It illustrates how you can supply the entire SQL statement of an xsql:query action handler as a parameter. This technique is a powerful and beneficial tool when in the right hands, but if you deploy a similar page to your production system, then the user can execute any query that the database security privileges for the connection associated with the page allows. For example, the Adhoc Query Demo is set up to use a connection that maps to the scott account, so a user can query any data that scott would be allowed to query from SQL*Plus.

You can use the following techniques to make sure your pages are not abused:

- Make sure the database user account associated with the page has only the privileges for reading the tables and views you want your users to see.

- Use true bind variables instead of lexical bind variables when substituting single values in a SELECT statement. If you need to parameterize syntactic parts of your SQL statement, then lexical parameters are the only way to proceed. Otherwise, you should use true bind variables so that any attempt to pass an invalid value generates an error instead of producing an unexpected result.
Using the XSQL Pages Publishing Framework: Advanced Topics

This chapter discusses the following XSQL pages advanced topics:

- Customizing the XSQL Configuration File Name
- Controlling How Stylesheets Are Processed
- Working with Array-Valued Parameters
- Setting Error Parameters on Built-in Actions
- Including XMLType Query Results in XSQL Pages
- Handling Posted XML Content
- Producing PDF Output with the FOP Serializer
- Performing XSQL Customizations

Customizing the XSQL Configuration File Name

By default, the XSQL pages framework expects the configuration file to be named XSQLConfig.xml. When moving between development, test, and production environments, you can switch between different versions of an XSQL configuration file. To override the name of the configuration file read by the XSQL page processor, set the Java system property xsql.config.

The simplest technique is to specify a Java VM command-line flag such as -Dxsql.config=MyConfigFile.xml by defining a servlet initialization parameter named xsql.config. Add an <init-param> element to your web.xml file as part of the <servlet> tag that defines the XSQL Servlet as follows:

```xml
<servlet>
    <servlet-name>XSQL</servlet-name>
    <servlet-class>oracle.xml.xsql.XSQLServlet</servlet-class>
    <init-param>
        <param-name>xsql.config</param-name>
        <param-value>MyConfigFile.xml</param-value>
        <description>
            Please Use MyConfigFile.xml instead of XSQLConfig.xml
        </description>
    </init-param>
</servlet>
```
The servlet initialization parameter is only applicable to the servlet-based use of the XSQL engine. When using the XSQLCommandLine or XSQLRequest programmatic interfaces, use the System parameter instead.

---

**Note:** The configuration file is always read from the CLASSPATH. For example, if you specify a custom configuration parameter file named MyConfigFile.xml, then the XSQL processor attempts to read the XML file as a resource from the CLASSPATH. In a J2EE-style servlet environment, you must place your MyConfigFile.xml in the .\WEB-INF\classes directory (or some other top-level directory that will be found on the CLASSPATH). If both the servlet initialization parameter and the System parameter are provided, then the servlet initialization parameter value is used.

---

**Controlling How Stylesheets Are Processed**

This section contains the following topics:

- Overriding Client Stylesheets
- Controlling the Content Type of the Returned Document
- Assigning the Stylesheet Dynamically
- Processing XSLT Stylesheets in the Client
- Providing Multiple Stylesheets

**Overriding Client Stylesheets**

If the current XSQL page being requested allows it, then you can supply an XSLT stylesheet URL in the request. This technique enables you to either override the default stylesheet or apply a stylesheet where none is applied by default. The client-initiated stylesheet URL is provided by supplying the xml-stylesheet parameter as part of the request. The valid values for this parameter are the following:

- Any relative URL interpreted relative to the XSQL page being processed.
- Any absolute URL that uses the HTTP protocol scheme, provided it references a trusted host as defined in the XSQL configuration file.
- The literal value none. Setting xml-stylesheet=none is useful during development to temporarily "short-circuit" the XSLT stylesheet processing to determine what XML datagram your stylesheet is seeing. Use this technique to determine why a stylesheet is not producing expected results.

You can allow client override of stylesheets for an XSQL page in the following ways:

- Setting the allow-client-style configuration parameter to no in the XSQL configuration file
- Explicitly including an allow-client-style="no" attribute on the document element of any XSQL page

If client-override of stylesheets has been globally disabled by default in the XSQL configuration file, any page can still enable client-override explicitly by including an allow-client-style="yes" attribute on the document element of that page.
Controlling the Content Type of the Returned Document

Setting the content type of the data that you serve enables the requesting client to correctly interpret the data that you return. If your stylesheet uses an `<xsl:output>` element, then the XSQL processor infers the media type and encoding of the returned document from the media-type and encoding attributes of `<xsl:output>`.

The stylesheet in Example 15–1 uses the `media-type="application/vnd.ms-excel"` attribute on `<xsl:output>`. This instruction transforms the results of an XSQL page containing a standard query of the hr.employees table into Microsoft Excel format.

Example 15–1  empToExcel.xsl

```xml
```

The following XSQL page makes use of the stylesheet in Example 15–1:

```xml
```

Assigning the Stylesheet Dynamically

If you include an `<xsl:stylesheet>` instruction at the top of your .xsql file, then the XSQL page processor considers it for use in transforming the resulting XML datagram. Consider the emp_test.xsql page shown in Example 15–2.

Example 15–2  emp_test.xsql

```xml
The page in Example 15–2 uses the emp.xsl stylesheet to transform the results of the employees query in the server tier before returning the response to the requestor. The processor accesses the stylesheet by the URL provided in the href pseudo-attribute on the <?xml-stylesheet?> processing instruction.

Suppose that you want to change XSLT stylesheets dynamically based on arguments passed to the XSQL servlet. You can achieve this goal by using a lexical parameter in the href attribute of your xml-stylesheet processing instruction, as shown in the following sample instruction:

```xml
<?xml-stylesheet type="text/xsl" href="{@filename}.xsl"?>
```

You can then pass the value of the filename parameter as part of the URL request to XSQL servlet.

You can also use the <xsql:set-page-param> element in an XSQL page to set the value of the parameter based on a SQL query. For example, the XSQL page in Example 15–3 selects the name of the stylesheet to use from a table by assigning the value of a page-private parameter.

**Example 15–3  emp_test_dynamic.xsql**

```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="{@sheet}.xsl"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
  <xsql:set-page-param bind-params="UserCookie" name="sheet">
    SELECT stylesheet_name
    FROM user_prefs
    WHERE username = ?
  </xsql:set-page-param>
  <xsql:query>
    SELECT *
    FROM employees
    ORDER BY salary DESC
  </xsql:query>
</page>
```

**Processing XSLT Stylesheets in the Client**

Some browsers support processing XSLT stylesheets in the client. These browsers recognize the stylesheet to be processed for an XML document by using an <?xml-stylesheet?> processing instruction. The use of <?xml-stylesheet?> for this purpose is part of the W3C Recommendation from June 29, 1999 entitled “Associating Stylesheets with XML Documents, Version 1.0”.

By default, the XSQL pages processor performs XSLT transformations in the server. By adding client="yes" to your <?xml-stylesheet?> processing instruction in your XSQL page, however, you can defer XSLT processing to the client. The processor serves the XML datagram "raw" with the current <?xml-stylesheet?> element at the top of the document.

**Providing Multiple Stylesheets**

You can include multiple <?xml-stylesheet?> processing instructions at the top of an XSQL page. The instructions can contain an optional media pseudo-attribute. If specified, the processor case-insensitively compares the value of the media pseudo-attribute with the value of the User-Agent string in the HTTP header. If the value of the media pseudo-attribute matches part of the User-Agent string, then the processor selects the current <?xml-stylesheet?> instruction for use. Otherwise, the
processor ignores the instruction and continues looking. The processor uses the first matching processing instruction in document order. An instruction without a media pseudo-attribute matches all user agents.

Example 15–4 shows multiple processing instructions at the top of an XSQL file. The processor uses doyouxml-lynx.xsl for Lynx browsers, doyouxml-ie.xsl for Internet Explorer 5.0 or 5.5 browsers, and doyouxml.xsl for all others.

Example 15–4  Multiple <?xml-stylesheet ?> Processing Instructions

```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" media="lynx" href="doyouxml-lynx.xsl" ?>
<?xml-stylesheet type="text/xsl" media="msie 5" href="doyouxml-ie.xsl" ?>
<?xml-stylesheet type="text/xsl" href="doyouxml.xsl" ?>
```

Table 15–1 summarizes the supported pseudo-attributes allowed on the <?xml-stylesheet ?> processing instruction.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type = &quot;string&quot;</td>
<td>Indicates the MIME type of the associated stylesheet. For XSLT stylesheets, this attribute must be set to the string text/xsl. This attribute may be present or absent when using the serializer attribute, depending on whether an XSLT stylesheet has to execute before invoking the serializer, or not.</td>
</tr>
<tr>
<td>href = &quot;URL&quot;</td>
<td>Indicates the relative or absolute URL to the XSLT stylesheet to be used. If an absolute URL is supplied that uses the http protocol scheme, the IP address of the resource must be a trusted host listed in the XSQL configuration file (by default, named XSQLConfig.xml).</td>
</tr>
<tr>
<td>media = &quot;string&quot;</td>
<td>Performs a case-insensitive match on the User-Agent string from the HTTP header sent by the requesting device. This attribute is optional. The current &lt;?xml-stylesheet ?&gt; processing instruction is used only if the User-Agent string contains the value of the media attribute; otherwise it is ignored.</td>
</tr>
<tr>
<td>client = &quot;boolean&quot;</td>
<td>Defers the processing of the associated XSLT stylesheet to the client if set to yes. The raw XML datagram is sent to the client with the current &lt;?xml-stylesheet ?&gt; instruction at the top of the document. The default if not specified is to perform the transformation in the server.</td>
</tr>
</tbody>
</table>
| serializer = "string" | By default, the XSQL page processor uses the following:
- XML DOM serializer if no XSLT stylesheet is used
- XSLT processor serializer if an XSLT stylesheet is used
Specifying this pseudo-attribute indicates that a custom serializer implementation must be used instead.
Valid values are either the name of a custom serializer defined in the <serializerdefs> section of the XSQL configuration file or the string java:fully.qualified.Classname. If both an XSLT stylesheet and the serializer attribute are present, then the processor performs the XSLT transformation first, then invokes the custom serializer to render the final result to the OutputStream or PrintWriter. |

Working with Array-Valued Parameters

This section contains the following topics:

- Supplying Values for Array-Valued Parameters
- Setting Array-Valued Page or Session Parameters from Strings
Supplied Values for Array-Valued Parameters

Request parameters, session parameters, and page-private parameters can have arrays of strings as values. To treat the value of a parameter as an array, add two empty square brackets to the end of its name. For example, if an HTML form is posted with four occurrences of a input control named productid, then use the notation productid[] to refer to the array-valued productid parameter. If you refer to an array-valued parameter without using the array-brackets notation, then the XSQL processor uses the value of the first array entry.

Note: The XSQL processor does not support use of numbers inside the array brackets. That is, you can refer to productid or productid[], but not productid[2].

Suppose that you refer to an array-valued parameter as a lexical substitution parameter inside an action handler attribute value or inside the content of an action handler element. The XSQL page processor converts its value to a comma-delimited list of non-null and non-empty strings in the order that they exist in the array. Example 15–5 shows an XSQL page with an array-valued parameter.

Example 15–5 Using an Array-Valued Parameter in an XSQL Page

```
<page xmlns:xsql="urn:oracle-xsql">
  <xsql:query>
    SELECT description
    FROM product
    WHERE productid in ( {$productid[]} ) /* Using lexical parameter */
  </xsql:query>
</page>
```

You can invoke the XSQL command-line utility to supply multiple values for the productid parameter in Page.xsql as follows:

```
xsql Page.xsql productid=111 productid=222 productid=333 productid=444
```

The preceding command sets the productid[] array-valued parameter to the value \"111\",\"222\",\"333\",\"444\". The XSQL page processor replaces the {productid[]} expression in the query with the string "111,222,333,444".

Note that you can also pass multi-valued parameters programmatically through the XSQLRequest API, which accepts a java.util.Dictionary of named parameters. You can use a Hashtable and call its put(name, value) method to add String-valued parameters to the request. To add multi-valued parameters, put a value of type String[] instead of type String.

Note: Only request parameters, page-private parameters, and session parameters can use string arrays. The <xsql:set-stylesheet-param> and <xsql:set-cookie> actions only support working with parameters as simple string values. To refer to a multi-valued parameter in your XSLT stylesheet, use <xsql:include-param> to include the multi-valued parameter into your XSQL datapage, then use an XPath expression in the stylesheet to refer to the values from the datapage.
Setting Array-Valued Page or Session Parameters from Strings

You can set the value of a page-private parameter or session parameter to a string-array value by using the array brackets notation on the name as follows:

<!-- param name contains array brackets -->
<xsql:set-page-param name="names[]" value="Tom Jane Joe"/>

You set the value similarly for session parameters, as shown in the following example:

<xsql:set-session-param name="dates[]" values="12-APR-1962 15-JUL-1968"/>

By default, when the name of the parameter uses array brackets, the XSQL processor treats the value as a space-or-comma-delimited list and tokenizes it.

The resulting string array value contains these separate tokens. In the preceding examples, the names[] parameter is the string array ["Tom", "Jane", "Joe"] and the dates[] parameter is the string array ["12-APR-1962", "15-JUL-1968"].

To handle strings that contain spaces, the tokenization algorithm first checks the string for the presence of commas. If at least one comma is found in the string, then commas are used as the token delimiter. For example, the following action sets the value of the names[] parameter to the string array ["Tom Jones", "Jane York"]:

<!-- param name contains array brackets -->
<xsql:set-page-param name="names[]" value="Tom Jones,Jane York"/>

By default, when you set a parameter whose name does not end with the array-brackets, then the string-tokenization does not occur. Thus, the following action sets the parameter names to the literal string "Tom Jones,Jane York":

<!-- param name does NOT contain array brackets -->
<xsql:set-page-param name="names" values="Tom Jones,Jane York"/>

You can force the string to be tokenized by including the treat-list-as-array="yes" attribute on the <xsql:set-page-param> or <xsql:set-session-param> actions. When this attribute is set, the XSQL processor assigns a comma-delimited string of the tokenized values to the parameter. For example, the following action sets the parameter names to the literal string "Tom,Jane,Joe":

<!-- param name does NOT contain array brackets -->
<xsql:set-page-param name="names" values="Tom Jane Joe" treat-list-as-array="yes"/>

When you are setting the value of a simple string-valued parameter and you are tokenizing the value with treat-list-as-array="yes", you can include the quote-array-values="yes" attribute to surround the comma-delimited values with single-quotes. Thus, the following action assigns the literal string value "'Tom Jones','Jane York','Jimmy'" to the names parameter:

<!-- param name does NOT contain array brackets -->
<xsql:set-page-param name="names" values="Tom Jones,Jane York,Jimmy" treat-list-as-array="yes" quote-array-values="yes"/>

Binding Array-Valued Parameters in SQL and PL/SQL Statements

Where string-valued scalar bind variables are supported in an XSQL page, you can also bind array-valued parameters. Use the array parameter name, for example, myparam[], in the list of parameter names that you supply for the bind-params
attribute. This technique enables you to process array-valued parameters in SQL statements and PL/SQL procedures.

The XSQL processor binds array-valued parameters as a nested table object type named XSQL_TABLE_OF_VARCHAR. You must create this type in your current schema with the following DDL statement:

```
CREATE TYPE xsq_table_of_varchar AS TABLE OF VARCHAR2(2000);
```

Although the type must have the name xsq_table_of_varchar, you can change the dimension of the VARCHAR2 string if desired. Of course, you have to make it as long as any string value you expect to handle in your array-valued string parameters.

Consider the PL/SQL function shown in Example 15–6.

**Example 15–6 testTableFunction**

```
FUNCTION testTableFunction(p_name  XSQL_TABLE_OF_VARCHAR,
                           p_value XSQL_TABLE_OF_VARCHAR)
RETURN VARCHAR2 IS
  lv_ret     VARCHAR2(4000);
  lv_numElts INTEGER;
BEGIN
  IF p_name IS NOT NULL THEN
    lv_numElts := p_name.COUNT;
    FOR j IN 1..lv_numElts LOOP
      IF (j > 1) THEN
        lv_ret := lv_ret||':';
      END IF;
      lv_ret := lv_ret||p_name(j)||'='||p_value(j);
    END LOOP;
  END IF;
  RETURN lv_ret;
END;
```

The XSQL page in Example 15–7 shows how to bind two array-valued parameters in a SQL statement that uses testTableFunction.

**Example 15–7 XSQL Page with Array-Valued Parameters**

```
<page xmlns:xsql="urn:oracle-xsql" connection="demo"
     someNames="aa,bb,cc" someValues="11,22,33">
  <xsql:query bind-params="someNames[] someValues[]">  
    SELECT testTableFunction(?,?) AS example  
    FROM dual  
  </xsql:query>
</page>
```

Executing the XSQL page in Example 15–7 generates the following datagram:

```
<page someNames="aa,bb,cc" someValues="11,22,33">
  <ROWSET>
    <ROW num="1">
      <EXAMPLE>aa=11:bb=22:cc=33</EXAMPLE>
    </ROW>
  </ROWSET>
</page>
```

This technique shows that the XSQL processor bound the array-valued someNames[] and someValues[] parameters as table collection types. It iterated over the values and
concatenated them to produce the "aa=11:bb=22:cc=33" string value as the return value of the PL/SQL function.

You can mix any number of regular parameters and array-valued parameters in your bind-params string. Use the array-bracket notation for the parameters that you want to be bound as arrays.

**Note:** If you run the page in Example 15–7 but you have not created the XSQL_TABLE_OF_VARCHAR type as illustrated earlier, then you receive an error such as the following:

```xml
<page someNames="aa,bb,cc" someValues="11,22,33">
  <xsql-error code="17074" action="xsql:query">
    <statement>
      select testTableFunction(?,?) as example from dual
    </statement>
    <message>
      invalid name pattern: SCOTT.XSQL_TABLE_OF_VARCHAR
    </message>
  </xsql-error>
</page>
```

Because the XSQL processor binds array parameters as nested table collection types, you can use the `TABLE()` operator with the `CAST()` operator in SQL to treat the nested table bind variable value as a table of values. You can then query this table. This technique is especially useful in subqueries. The page in Example 15–8 uses an array-valued parameter containing employee IDs to restrict the rows queried from hr.employees.

**Example 15–8  Using an Array-Valued Parameter to Restrict Rows**

```xml
<page xmlns:xsql="urn:oracle-xsql" connection="hr">
  <xsql-set-page-param name="someEmployees[]" value="196,197"/>
  <xsql:query bind-params="someEmployees[]">
    SELECT first_name||' '||last_name AS name, salary
    FROM employees
    WHERE employee_id IN {
      SELECT * FROM TABLE(CAST( ? AS xsql_table_of_varchar))
    }
  </xsql:query>
</page>
```

The XSQL page in Example 15–8 generates a datagram such as the following:

```xml
<ROWSET>
  <ROW num="1">
    <NAME>Alana Walsh</NAME>
    <SALARY>3100</SALARY>
  </ROW>
  <ROW num="2">
    <NAME>Kevin Feeny</NAME>
    <SALARY>3000</SALARY>
  </ROW>
</ROWSET>
```
Example 15–7 and Example 15–8 show how to use bind-params with <xsql:query>, but these techniques work for <xsql:dml>, <xsql:include-owa>, <xsql:ref-cursor-function>, and other actions that accept SQL or PL/SQL statements.

Note that PL/SQL index-by tables work with the OCI JDBC driver but not the JDBC thin driver. By using the nested table collection type XSQL_TABLE_OF_VARCHAR, you can use array-valued parameters with either driver. In this way you avoid losing the programming flexibility of working with array values in PL/SQL.

**Setting Error Parameters on Built-in Actions**

The XSQL page processor determines whether an action encountered a non-fatal error during its execution. For example, an attempt to insert a row or call a stored procedure can fail with a database exception that will get included in your XSQL data page as an <xsql-error> element.

You can set a page-private parameter in a built-in XSQL action when the action reports a nonfatal error. Use the error-param attribute on the action to enable this feature. For example, to set the parameter "dml-error" when the statement inside the <xsql:dml> action encounters a database error, you can use the technique shown in Example 15–9.

**Example 15–9 Setting an Error Parameter**

```
<xsql:dml error-param="dml-error" bind-params="val">
  INSERT INTO yourtable(somecol)
  VALUES(?)
</xsql:dml>
```

If the execution of the <xsql:dml> action encounters an error, then the XSQL processor sets the page-private parameter dml-error to the string "Error". If the execution is successful, then the XSQL processor does not assign a value to the error parameter. In Example 15–9, if the page-private parameter dml-error already exists, then it retains its current value. If it does not exist, then it continues not to exist.

**Using Conditional Logic with Error Parameters**

By using the error parameter in combination with <xsql:if-param>, you can achieve conditional behavior in your XSQL page template. For example, assume that your connection definition sets the AUTOCOMMIT flag to false on the connection named demo in the XSQL configuration file. The XSQL page shown in Example 15–10 illustrates how you might roll back the changes made by a previous action if a subsequent action encounters an error.

**Example 15–10 Achieving Conditional Behavior with an Error Parameter**

```
<!-- NOTE: Connection 'demo' must not set to autocommit! -->
<page connection='demo' xmlns:xsql='urn:oracle-xsql'>
  <xsql:dml error-param="dml-error" bind-params="val">
    INSERT INTO yourtable(somecol)
    VALUES(?)
  </xsql:dml>
  <!-- This second statement will commit if it succeeds -->
  <xsql:dml commit="yes" error-param="dml-error" bind-params="val2">
    INSERT INTO anothertable(anothercol)
    VALUES(?)
  </xsql:dml>
  <xsql:if-param name="dml-error" exists="yes">
    <!-- Code to roll back changes here -->
  </xsql:if-param>
</page>
```
If you have written custom action handlers, and if your custom actions call
reportMissingAttribute(), reportError(), or reportErrorIncludingStatement() to
report non-fatal action errors, then they automatically pick up this feature as well.

Formatting XSQL Action Handler Errors

Errors raised by the processing of XSQL action elements are reported as XML elements
in a uniform way. This fact enables XSLT stylesheets to detect their presence and
optionally format them for presentation.

The action element in error is replaced in the page by the following element:

```xml
<xsql-error action="xxx"/>
```

Depending on the error the `<xsql-error>` element contains:

- A nested `<message>` element
- A `<statement>` element with the offending SQL statement

Example 15–11 shows an XSLT stylesheet that uses this information to display error
information on the screen.

```xml
Example 15–11  XSLTStylesheet
<xsl:if test="/xsql-error">
  <table style="background:yellow">
    <xsl:for-each select="/xsql-error">
      <tr>
        <td><b>Action</b></td>
        <td><xsl:value-of select="@action"/></td>
      </tr>
      <tr valign="top">
        <td><b>Message</b></td>
        <td><xsl:value-of select="message"/></td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:if>
```

Including XMLType Query Results in XSQL Pages

Oracle Database supports XMLType for storing and querying XML-based database
content. You can exploit database XML features to produce XML for inclusion in your
XSQL pages by using one of the following techniques:

- `<xsql:query>` handles any query including columns of type XMLType, but it handles
  XML markup in CLOB and VARCHAR2 columns as literal text.
- `<xsql:include-xml>` parses and includes a single CLOB or string-based XML
document retrieved from a query.

One difference between the preceding approaches is that `<xsql:include-xml>` parses
the literal XML appearing in a CLOB or string value on the fly to turn it into a tree of
elements and attributes. In contrast, `<xsql:query>` leaves XML markup in CLOB or
string-valued columns as literal text.
Another difference is that while `<xsql:query>` can handle query results of any number of columns and rows, `<xsql:include-xml>` works on a single column of a single row. Accordingly, when using `<xsql:include-xml>`, the SELECT statement inside it returns a single row containing a single column. The column can either be a CLOB or a VARCHAR2 value containing a well-formed XML document. The XSQL engine parses the XML document and includes it in your XSQL page.

Example 15–12 uses nested `XmlAgg()` functions to aggregate the results of a dynamically-constructed XML document containing departments and nested employees. The functions aggregate the document into a single "result" document wrapped in a `<DepartmentList>` element.

**Example 15–12  Aggregating a Dynamically-Constructed XML Document**

```xml
<xsql:query connection="hr" xmlns:xsql="urn:oracle-xsql">
  SELECT XmlElement('DepartmentList',
    XmlAgg{
      XmlElement("Department",
        XmlAttributes(department_id AS "Id"),
        XmlForest(department_name AS "Name"),
        (SELECT XmlElement("Employees",
          XmlAgg{
            XmlElement("Employee",
              XmlAttributes(employee_id AS "Id"),
              XmlForest(first_name||' '||last_name AS "Name",
                salary AS "Salary",
                job_id AS "Job")
            )
          )
        )
      )
    )
  )
FROM employees e
WHERE e.department_id = d.department_id
</xsql:query>
```

In another example, suppose you have a number of `<Movie>` XML documents stored in a table of `XMLType` called movies. Each document might look like the one shown in Example 15–13.

**Example 15–13  Movie XML Document**

```xml
<Movie Title="The Talented Mr.Ripley" RunningTime="139" Rating="R">
  <Director>
    <First>Anthony</First>
    <Last>Minghella</Last>
  </Director>
  <Cast>
    <Actor Role="Tom Ripley">
      <First>Matt</First>
      <Last>Damon</Last>
    </Actor>
    <Actress Role="Marge Sherwood">
      <First>Gwyneth</First>
      <Last>Paltrow</Last>
    </Actress>
</Movie>
```
<Actor Role="Dickie Greenleaf">
    <First>Jude</First>
    <Last>Law</Last>
    <Award From="BAFTA" Category="Best Supporting Actor"/>
</Actor>
</Cast>
</Movie>

You can use the built-in XPath query features to extract an aggregate list of all cast members who have received Oscar awards from any movie in the database. Example 15–14 shows a sample query.

**Example 15–14 Using XPath to Extract an Aggregate List**

```sql
SELECT XMLQUERY("<AwardedActors>
<Actor>...</Actor>
<Actress>...</Actress>
</AwardedActors>
FROM movies m
```

To include this query result of XMLType in your XSQL page, paste the query inside an `<xsql:query>` element. Make sure you include an alias for the query expression, as shown in Example 15–15.

**Example 15–15 Including an XMLType Query Result**

```sql
<xsql:query connection="demo" xmlns:xsql="urn:oracle-xsql">
    SELECT XMLQUERY("<AwardedActors>
    <Actor>...</Actor>
    <Actress>...</Actress>
    </AwardedActors>
    FROM movies m
</xsql:query>
```

You can use the combination of `XmlElement()` and `XmlAgg()` to make the database aggregate all of the XML fragments identified by the query into single, well-formed XML document. The functions work together to produce a well-formed result like the following:

```xml
<AwardedActors>
    <Actor>...</Actor>
    <Actress>...</Actress>
</AwardedActors>
```

You can use the standard XSQL bind variable capabilities in the middle of an XPath expression if you concatenate the bind variable into the expression. For example, to parameterize the value Oscar into a parameter named `award-from`, you can use an XSQL page like the one shown in Example 15–16.

**Example 15–16 Using XSQL Bind Variables in an XPath Expression**

```sql
<xsql:query connection="orcl92" xmlns:xsql="urn:oracle-xsql">
    /* Using a bind variable in an XPath expression */
    SELECT XMLQUERY("<AwardedActors>
        <Actor>...</Actor>
        <Actress>...</Actress>
    </AwardedActors>
    FROM movies m
</xsql:query>
```
Handling Posted XML Content

In addition to simplifying the assembly and transformation of XML content, the XSQL pages framework enables you to handle posted XML content. Built-in actions provide the following advantages:

- Simplify the handling of posted data from both XML document and HTML forms
- Enable data to be posted directly into a database table by using XSU

XSU can perform database inserts, updates, and deletes based on the content of an XML document in canonical form for a target table or view. For a specified table, the canonical XML form of its data is given by one row of XML output from a `SELECT *` query. When given an XML document in this form, XSU can automate the DML operation.

By combining XSU with XSLT, you can transform XML in any format into the canonical format expected by a given table. XSU can then perform DML on the resulting canonical XML.

The following built-in XSQL actions make it possible for you to exploit this capability from within your XSQL pages:

- `<xsql:insert-request>`
  Insert the optionally transformed XML document that was posted in the request into a table.

- `<xsql:update-request>`
  Update the optionally transformed XML document that was posted in the request into a table or view.

- `<xsql:delete-request>`
  Delete the optionally transformed XML document that was posted in the request from a table or view.

- `<xsql:insert-param>`
  Insert the optionally transformed XML document that was posted as the value of a request parameter into a table or view.

If you target a database view with your insert, then you can create `INSTEAD OF INSERT` triggers on the view to further automate the handling of the posted information. For example, an `INSTEAD OF INSERT` trigger on a view can use PL/SQL to check for the existence of a record and intelligently choose whether to do an `INSERT` or an `UPDATE` depending on the result of this check.

Understanding XML Posting Options

The XSQL pages framework can handle posted data in the following scenarios:

- A client program sends an HTTP POST message that targets an XSQL page. The request body contains an XML document; the HTTP header reports a `ContentType` of "text/xml".
  In this case, `<xsql:insert-request>,<xsql:update-request>, or `<xsql:delete-request>` can insert, update, or delete the content of the posted XML in the target table. If you transform the posted XML with XSLT, then the posted document is the source for the transformation.

- A client program sends an HTTP GET request for an XSQL page, one of whose parameters contains an XML document.
In this case, you can use the `<xsql:insert-param>` action to insert the content of the posted XML parameter value in the target table. If you transform the posted XML document with XSLT, then the XML document in the parameter value is the source document for this transformation.

- A browser submits an HTML form with `method="POST"` whose action targets an XSQL page. The request body of the HTTP POST message contains an encoded version of the form fields and values with a `ContentType` of "application/x-www-form-urlencoded".

In this case, the request does not contain an XML document, but an encoded version of the form parameters. To make all three of these cases uniform, however, the XSQL page processor materializes on demand an XML document from the form parameters, session variables, and cookies contained in the request. The XSLT processor transforms this dynamically-materialized XML document into canonical form for DML by using `<xsql:insert>`, `<xsql:update-request>`, or `<xsql:delete-request>`.

When working with posted HTML forms, the dynamically materialized XML document has the form shown in Example 15–17.

**Example 15–17  XML Document Generated from HTML Form**

```
<request>
  <parameters>
    <firstparamname>firstparamvalue</firstparamname>
    ...
    <lastparamname>lastparamvalue</lastparamname>
  </parameters>
  <session>
    <firstsessionparamname>firstsessionparamvalue</firstsessionparamname>
    ...
    <lastsessionparamname>lastsessionparamvalue</lastsessionparamname>
  </session>
  <cookies>
    <firstcookie>firstcookievalue</firstcookiename>
    ...
    <lastcookie>firstcookievalue</lastcookiename>
  </cookies>
</request>
```

If multiple parameters are posted with the same name, then the XSQL processor automatically creates multiple `<row>` elements to make subsequent processing easier. Assume that a request posts or includes the following parameters and values:

- `id = 101`
- `name = Steve`
- `id = 102`
- `name = Sita`
- `operation = update`

The XSQL page processor creates a set of parameters as follows:

```
<request>
  <parameters>
    <row>
      <id>101</id>
      <name>Steve</name>
    </row>
  </parameters>
```
You need to provide an XSLT stylesheet that transforms this materialized XML document containing the request parameters into canonical format for your target table. Thus, you can build an XSQL page as follows:

```xsql
<!-- ShowRequestDocument.xsql
| Show Materialized XML Document for an HTML Form
-->
<xsql:include-request-params xmlns:xsql="urn:oracle-xsql"/>
```

With this page in place, you can temporarily modify your HTML form to post to the ShowRequestDocument.xsql page. In the browser you will see the "raw" XML for the materialized XML request document, which you can save and use to develop the XSL transformation.

## Producing PDF Output with the FOP Serializer

Using the XSQL pages framework support for custom serializers, the oracle.xml.xsql.serializers.XSQLFOPSerializer class provides integration with the Apache FOP processor. The FOP processor renders a PDF document from an XML document containing XSL Formatting Objects.

As described in Table 14–1, the demo directory includes the emptablefo.xsl stylesheet and emptable.xsql page as illustrations. If you get an error trying to use the FOP serializer, then probably you do not have all of the required JAR files in the CLASSPATH. The XSQLFOPSerializer class resides in the separate xml.jar file, which must be included in the CLASSPATH to use the FOP integration. You also need to add the following additional Java archives to your CLASSPATH:

- fop.jar - from Apache, version 0.20.3 or higher
- batik.jar - from the FOP distribution
-avalon-framework-4.0.jar - from FOP distribution
- logkit-1.0.jar - from FOP distribution

In case you want to customize the implementation, the source code for the FOP serializer provided in this release is shown in Example 15–18.

### Example 15–18  Source Code for FOP Serializer

```java
package oracle.xml.xsql.serializers;
import org.w3c.dom.Document;
import org.apache.log.Logger;
import org.apache.log.Hierarchy;
import org.apache.fop.messaging.MessageHandler;
import org.apache.log.LogTarget;
import oracle.xml.xsql.XSQLPageRequest;
import oracle.xml.xsql.XSQLDocumentSerializer;
import org.apache.fop.apps.Driver;
import org.apache.log.output.NullOutputLogTarget;
```
/**
 * Tested with the FOP 0.20.3RC release from 19-Jan-2002
 */
public class XSQLFOPSerializer implements XSQLDocumentSerializer {
    private static final String PDFMIME = "application/pdf";
    public void serialize(Document doc, XSQLPageRequest env) throws Throwable {
        try {
            // First make sure we can load the driver
            Driver FOPDriver = new Driver();
            // Tell FOP not to spit out any messages by default.
            // You can modify this code to create your own FOP Serializer
            // that logs the output to one of many different logger targets
            // using the Apache LogKit API
            Logger logger=Hierarchy.getDefaultHierarchy().getLoggerFor("XSQLServlet");
            logger.setLogTargets(new LogTarget[]{new NullOutputLogTarget()});
            FOPDriver.setLogger(logger);
            // Some of FOP's messages appear to still use MessageHandler.
            MessageHandler.setOutputMethod(MessageHandler.NONE);
            // Then set the content type before getting the reader
            env.setContentType(PDFMIME);
            FOPDriver.setOutputStream(env.getOutputStream());
            FOPDriver.setRenderer(FOPDriver.RENDER_PDF); FOPDriver.render(doc);
        }
        catch (Exception e) {
            // Cannot write PDF output for the error anyway.
            // So maybe this stack trace will be useful info
            e.printStackTrace(System.err);
        }
    }
}

See Also:  http://xml.apache.org/fop to learn about the Formatting Objects Processor

Performing XSQL Customizations

This section contains the following topics:

- Writing Custom XSQL Action Handlers
- Implementing Custom XSQL Serializers
- Using a Custom XSQL Connection Manager for JDBC Datasources
- Writing Custom XSQL Connection Managers
- Implementing a Custom XSQLErrorHandler
- Providing a Custom XSQL Logger Implementation

Writing Custom XSQL Action Handlers

When a task requires custom processing, and none of the built-in actions listed in Table 30–2, " XSQL Configuration File Settings" does exactly what you need, you can write your own actions. The XSQL pages engine processes an XSQL page by looking for action elements from the xsql namespace and invoking an appropriate action element handler class to process each action. The processor supports any action that implements the XSQLActionHandler interface. All of the built-in actions implement this interface.
The XSQL engine processes the actions in a page in the following way. For each action in the page, the engine performs the following steps:

1. Constructs an instance of the action handler class using the default constructor
2. Initializes the handler instance with the action element object and the page processor context by invoking the method `init(Element actionElt, XSQLPageRequest context)`
3. Invokes the method that allows the handler to handle the action `handleAction(Node result)`

For built-in actions, the engine can map the XSQL action element name to the Java class that implements the handler of the action. Table 30–2, "XSQL Configuration File Settings" lists the built-in actions and their corresponding classes.

For user-defined actions, use the following built-in action, replacing `fully.qualified.Classname` with the name of your class:

```xml
<xsql:action handler='fully.qualified.Classname' ... />
```

The `handler` attribute provides the fully-qualified name of the Java class that implements the custom action handler.

**Implementing the XSQLActionHandler Interface**

To create a custom action handler, provide a class that implements the `oracle.xml.xsql.XSQLActionHandler` interface. Most custom action handlers extend `oracle.xml.xsql.XSQLActionHandlerImpl`, which provides a default implementation of the `init()` method and offers useful helper methods.

When an action handler's `handleAction()` method is invoked by the XSQL pages processor, a DOM fragment is passed to the action implementation. The action handler appends any dynamically created XML content returned to the page to the root node.

The XSQL processor conceptually replaces the action element in the XSQL page with the content of this document fragment. It is legal for an action handler to append nothing to this fragment if it has no XML content to add to the page.

While writing custom action handlers, some methods on the `XSQLActionHandlerImpl` class are helpful. Table 15–2 lists these methods.
### Table 15–2 Helpful Methods in the XSQLActionHandlerImpl Class

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getActionElement</code></td>
<td>Returns the current action element being handled.</td>
</tr>
<tr>
<td><code>getActionElementContent</code></td>
<td>Returns the text content of the current action element, with all lexical parameters substituted appropriately.</td>
</tr>
<tr>
<td><code>getPageRequest</code></td>
<td>Returns the current XSQL pages processor context. Using this object you do the following:</td>
</tr>
<tr>
<td></td>
<td>■ <code>setPageParam()</code></td>
</tr>
<tr>
<td></td>
<td>Set a page parameter value.</td>
</tr>
<tr>
<td></td>
<td>■ <code>getPostedDocument()</code>/<code>setPostedDocument()</code></td>
</tr>
<tr>
<td></td>
<td>Get or set the posted XML document.</td>
</tr>
<tr>
<td></td>
<td>■ <code>translateURL()</code></td>
</tr>
<tr>
<td></td>
<td>Translate a relative URL to an absolute URL.</td>
</tr>
<tr>
<td></td>
<td>■ <code>getRequestObject()</code>/<code>setRequestObject()</code></td>
</tr>
<tr>
<td></td>
<td>Get or set objects in the page request context that can be shared across actions in a single page.</td>
</tr>
<tr>
<td></td>
<td>■ <code>getJDBCConnection()</code></td>
</tr>
<tr>
<td></td>
<td>Gets the JDBC connection in use by this page (possible null if no connection in use).</td>
</tr>
<tr>
<td></td>
<td>■ <code>getRequestType()</code></td>
</tr>
<tr>
<td></td>
<td>Detect whether you are running in the Servlet, Command Line, or Programmatic context. For example, if the request type is Servlet then you can cast the XSQLPageRequest object to the more specific XSQLServletPageRequest to access servlet-specific methods such as <code>getHttpServletRequest</code>, <code>getHttpServletResponse</code>, and <code>getServletContext</code>.</td>
</tr>
<tr>
<td><code>getAttributeAllowingParam</code></td>
<td>Retrieves the attribute value from an element, resolving any XSQL lexical parameter references that might appear in value of the attribute. Typically this method is applied to the action element itself, but it is also useful for accessing attributes of subelements. To access an attribute value without allowing lexical parameters, use the standard <code>getAttribute()</code> method on the DOM Element interface.</td>
</tr>
<tr>
<td><code>appendSecondaryDocument</code></td>
<td>Appends the contents of an external XML document to the root of the action handler result content.</td>
</tr>
<tr>
<td><code>addResultElement</code></td>
<td>Simplifies appending a single element with text content to the root of the action handler result content.</td>
</tr>
<tr>
<td><code>firstColumnOfFirstRow</code></td>
<td>Returns the first column value of the first row of a SQL statement. Requires the current page to have a connection attribute on its document element, or an error is returned.</td>
</tr>
<tr>
<td><code>getBindVariableCount</code></td>
<td>Returns the number of tokens in the space-delimited list of <code>bind-params</code>. This number indicates how many bind variables are expected to be bound to parameters.</td>
</tr>
<tr>
<td><code>handleBindVariables</code></td>
<td>Manages the binding of JDBC bind variables that appear in a prepared statement with the parameter values specified in the <code>bind-params</code> attribute on the current action element. If the statement is already using a number of bind variable slots in use as well, you can pass the number of existing bind variable slots in use as well. Managed by JDBC.</td>
</tr>
<tr>
<td><code>reportErrorIncludingStatement</code></td>
<td>Reports an error. The error includes the offending (SQL) statement that caused the problem and optionally includes a numeric error code.</td>
</tr>
<tr>
<td><code>reportFatalError</code></td>
<td>Reports a fatal error.</td>
</tr>
</tbody>
</table>
Performing XSQL Customizations

Example 15–19 shows a custom action handler named MyIncludeXSQLHandler that leverages one of the built-in action handlers. It uses arbitrary Java code to modify the XML fragment returned by this handler before appending its result to the XSQL page.

**Example 15–19**  

MyIncludeXSQLHandler.java

```java
import oracle.xml.xsql.*;
import oracle.xml.xsql.actions.XSQLIncludeXSQLHandler;
import org.w3c.dom.*;
import java.sql.SQLException;
public class MyIncludeXSQLHandler extends XSQLActionHandlerImpl {
    XSQLActionHandler nestedHandler = null;
    public void init(XSQLPageRequest req, Element action) {
        super.init(req, action);
        // Create an instance of an XSQLIncludeXSQLHandler and init() the handler by
        // passing the current request/action. This assumes the XSQLIncludeXSQLHandler
        // will pick up its href="xxx.xsql" attribute from the current action element.
        nestedHandler = new XSQLIncludeXSQLHandler();
        nestedHandler.init(req,action);
    }
    public void handleAction(Node result) throws SQLException {
        DocumentFragment df=result.getOwnerDocument().createDocumentFragment();
        nestedHandler.handleAction(df);
        // Custom Java code here can work on the returned document fragment
        // before appending the final, modified document to the result node.
        // For example, add an attribute to the first child.
        Element e = (Element)df.getFirstChild();
        if (e != null) {
            e.setAttribute("ExtraAttribute","SomeValue");
        }
        result.appendChild(df);
    }
}
```

You may need to write custom action handlers that work differently based on whether the page is requested through the XSQL servlet, the XSQL command-line utility, or programmatically through the XSQLRequest class. You can invoke `getPageRequest()` in your action handler implementation to obtain a reference to the XSQLPageRequest interface for the current page request. By calling `getRequestType()` on the XSQLPageRequest object, you can determine whether the request is coming from the Servlet, Command Line, or Programmatic routes. If the return value is `Servlet`, then you can access the HTTP servlet request, response, and servlet context objects as shown in Example 15–20.

**Table 15–2  (Cont.)**  

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reportMissingAttribute</td>
<td>Reports an error that a required action handler attribute is missing by using the <code>&lt;xsql-error&gt;</code> element.</td>
</tr>
<tr>
<td>reportStatus</td>
<td>Reports action handler status by using the <code>&lt;xsql-status&gt;</code> element.</td>
</tr>
<tr>
<td>requiredConnectionProvided</td>
<td>Checks whether a connection is available for this request and outputs an errorgram into the page if no connection is available.</td>
</tr>
<tr>
<td>variableValue</td>
<td>Returns the value of a lexical parameter, taking into account all scoping rules that might determine its default value.</td>
</tr>
</tbody>
</table>

Example 15–19 shows a custom action handler named `MyIncludeXSQLHandler` that leverages one of the built-in action handlers. It uses arbitrary Java code to modify the XML fragment returned by this handler before appending its result to the XSQL page.

**Example 15–19**  

MyIncludeXSQLHandler.java

```java
import oracle.xml.xsql.*;
import oracle.xml.xsql.actions.XSQLIncludeXSQLHandler;
import org.w3c.dom.*;
import java.sql.SQLException;
public class MyIncludeXSQLHandler extends XSQLActionHandlerImpl {
    XSQLActionHandler nestedHandler = null;
    public void init(XSQLPageRequest req, Element action) {
        super.init(req, action);
        // Create an instance of an XSQLIncludeXSQLHandler and init() the handler by
        // passing the current request/action. This assumes the XSQLIncludeXSQLHandler
        // will pick up its href="xxx.xsql" attribute from the current action element.
        nestedHandler = new XSQLIncludeXSQLHandler();
        nestedHandler.init(req,action);
    }
    public void handleAction(Node result) throws SQLException {
        DocumentFragment df=result.getOwnerDocument().createDocumentFragment();
        nestedHandler.handleAction(df);
        // Custom Java code here can work on the returned document fragment
        // before appending the final, modified document to the result node.
        // For example, add an attribute to the first child.
        Element e = (Element)df.getFirstChild();
        if (e != null) {
            e.setAttribute("ExtraAttribute","SomeValue");
        }
        result.appendChild(df);
    }
}
```

You may need to write custom action handlers that work differently based on whether the page is requested through the XSQL servlet, the XSQL command-line utility, or programmatically through the XSQLRequest class. You can invoke `getPageRequest()` in your action handler implementation to obtain a reference to the XSQLPageRequest interface for the current page request. By calling `getRequestType()` on the XSQLPageRequest object, you can determine whether the request is coming from the Servlet, Command Line, or Programmatic routes. If the return value is `Servlet`, then you can access the HTTP servlet request, response, and servlet context objects as shown in Example 15–20.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reportMissingAttribute</td>
<td>Reports an error that a required action handler attribute is missing by using the <code>&lt;xsql-error&gt;</code> element.</td>
</tr>
<tr>
<td>reportStatus</td>
<td>Reports action handler status by using the <code>&lt;xsql-status&gt;</code> element.</td>
</tr>
<tr>
<td>requiredConnectionProvided</td>
<td>Checks whether a connection is available for this request and outputs an errorgram into the page if no connection is available.</td>
</tr>
<tr>
<td>variableValue</td>
<td>Returns the value of a lexical parameter, taking into account all scoping rules that might determine its default value.</td>
</tr>
</tbody>
</table>

Example 15–19 shows a custom action handler named `MyIncludeXSQLHandler` that leverages one of the built-in action handlers. It uses arbitrary Java code to modify the XML fragment returned by this handler before appending its result to the XSQL page.
**Example 15–20  Testing for the Servlet Request**

```java
XSQLServletPageRequest xspr = (XSQLServletPageRequest)getPageRequest();
if (xspr.getRequestType().equals("Servlet")) {
  HttpServletRequest req  = xspr.getHttpServletRequest();
  HttpServletResponse resp = xspr.getHttpServletResponse();
  ServletContext cont  = xspr.getServletContext();
  // Do something here with req, resp, or cont. Note that writing to the response
  // directly from a handler produces unexpected results. All the servlet or your
  // custom Serializer to write to the servlet response output stream at the right
  // moment later when all action elements have been processed.
}
```

**Using Multivalued Parameters in Custom XSQL Actions**

XSQLActionHandlerImpl is the base class for custom XSQL actions. It supports the following:

- Array-named lexical parameter substitution
- Array-named bind variables
- Simple-valued parameters

If your custom actions use methods such as `getAttributeAllowingParam()`, `getActionElementContent()`, or `handleBindVariables()` from this base class, you pick up multi-valued parameter functionality for free in your custom actions.

Use the `getParameterValues()` method on the `XSQLPageRequest` interface to explicitly get a parameter value as a `String[]`. The helper method `variableValues()` in `XSQLActionHandlerImpl` enables you to use this functionality from within a custom action handler if you need to do so programmatically.

**Implementing Custom XSQL Serializers**

You can implement a user-defined serializer class to control how the final XSQL datapage is serialized to a text or binary stream. A user-defined serializer must implement the `oracle.xml.xsql.XSQLDocumentSerializer` interface. The interface contains the following single method:

```java
void serialize(org.w3c.dom.Document doc, XSQLPageRequest env) throws Throwable;
```

Only DOM-based serializers are supported. A custom serializer class is expected to perform the following tasks in the correct order:

1. Set the content type of the serialized stream before writing any content to the output `PrintWriter` (or `OutputStream`).
   
   Set the type by calling `setContentType()` on the `XSQLPageRequest` passed to your serializer. When setting the content type, you can set a MIME type as follows:
   ```java
   env.setContentType("text/html");
   ```
   Alternatively, you can set a MIME type with an explicit output encoding character set as follows:
   ```java
   env.setContentType("text/html;charset=Shift_JIS");
   ```

2. Call either `getWriter()` or `getOutputStream()` (but not both) on the `XSQLPageRequest` to obtain the appropriate `PrintWriter` or `OutputStream` for serializing the content.
The custom serializer in Example 15–21 illustrates a simple implementation that serializes an HTML document containing the name of the document element of the current XSQL data page.

**Example 15–21  Custom Serializer**

```java
package oracle.xml.xsql.serializers;
import org.w3c.dom.Document;
import java.io.PrintWriter;
import oracle.xml.xsql.*;

public class XSQLSampleSerializer implements XSQLDocumentSerializer {
    public void serialize(Document doc, XSQLPageRequest env) throws Throwable {
        String encoding = env.getPageEncoding();  // Use same encoding as XSQL page
        // template. Set to specific
        // encoding if necessary
        String mimeType = "text/html"; // Set this to the appropriate content type
        // (1) Set content type using the setContentType on the XSQLPageRequest
        if (encoding != null && !encoding.equals("") ) {
            env.setContentType(mimeType+";charset="+encoding);
        } else {
            env.setContentType(mimeType);
        }
        // (2) Get the output writer from the XSQLPageRequest
        PrintWriter e = env.getWriter();
        // (3) Serialize the document to the writer
        e.println("<html>Document element is <b>"+
                    doc.getDocumentElement().getNodeName()+"</b></html>");
    }
}
```

**Techniques for Using a Custom Serializer**

There are two ways to use a custom serializer, depending on whether you need to first perform an XSLT transformation before serializing or not.

To perform an XSLT transformation before using a custom serializer, add the `serializer="java:fully.qualified.ClassName"` in the `<xml-stylesheet>` processing instruction at the top of your page. The following examples illustrates this technique:

```xml
<?xml version="1.0"?>
<xml-stylesheet type="text/xsl" href="mystyle.xsl"
    serializer="java:my.pkg.MySerializer"/>
```

If you only need the custom serializer, then leave out the `type` and `href` attributes. The following example illustrates this technique:

```xml
<?xml version="1.0"?>
<xml-stylesheet serializer="java:my.pkg.MySerializer"/>
```

**Assigning a Short Name to a Custom Serializer**

You can also assign a short name to your custom serializers in the `<serializerdefs>` section of the XSQL configuration file. You can then use the nickname in the serializer attribute instead to save typing. Note that the short name is case sensitive.

Assume that you have the information shown in Example 15–22 in your XSQL configuration file.
Performing XSQL Customizations

Example 15–22 Assigning Short Names to Custom Serializers

```xml
<XSQLConfig>
  <!--and so on. -->
  <serializerdefs>
    <serializer>
      <name>Sample</name>
      <class>oracle.xml.xsql.serializers.XSQLSampleSerializer</class>
    </serializer>
    <serializer>
      <name>FOP</name>
      <class>oracle.xml.xsql.serializers.XSQLFOPSerializer</class>
    </serializer>
  </serializerdefs>
</XSQLConfig>
```

You can use the short names “Sample” or “FOP” in a stylesheet instruction as follows:

```xml
<?xml-stylesheet type="text/xsl" href="emp-to-xslfo.xsl" serializer="FOP"/>
<?xml-stylesheet serializer="Sample"/>
```

The `XSQLPageRequest` interface supports both a `getWriter()` and a `getOutputStream()` method. Custom serializers can call `getOutputStream()` to return an `OutputStream` instance into which binary data can be serialized. When you use the XSQL servlet, writing to this output stream results in writing binary information to the servlet output stream.

The serializer shown in Example 15–23 illustrates an example of writing a dynamic GIF image. In this example the GIF image is a static "ok" icon, but it shows the basic technique that a more sophisticated image serializer needs to use.

Example 15–23 Writing a Dynamic GIF Image

```java
package oracle.xml.xsql.serializers;
import org.w3c.dom.Document;
import java.io.*;
import oracle.xml.xsql.*;
public class XSQLSampleImageSerializer implements XSQLDocumentSerializer {
  // Byte array representing a small *ok* GIF image
  private static byte[] okGif =
      { (byte)0x47,(byte)0x49,(byte)0x46,(byte)0x38,
        (byte)0x39,(byte)0x61,(byte)0xB,(byte)0x0,
        (byte)0x9,(byte)0x0,(byte)0xFFFF0080,(byte)0x0,
        (byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,
        (byte)0x00000080,(byte)0x000000FF,(byte)0x02C,
        (byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,
        (byte)0x0B,(byte)0x0,(byte)0x9,(byte)0x0,
        (byte)0x0,(byte)0x2,(byte)0x14,(byte)0xFFFFF8C,
        (byte)0xF,(byte)0xFFFFF97,(byte)0xFFFFF98,(byte)0xFFFFF9B,
        (byte)0x8,(byte)0xFFFFF9A2,(byte)0x79,(byte)0x0FFFFFE9,
        (byte)0xFFFFF85,(byte)0x7A,(byte)0x27,(byte)0xFFFFF91,
        (byte)0x5A,(byte)0xFFFFF9E3,(byte)0xFFFFF9EC,(byte)0x75,
        (byte)0x11,(byte)0xFFFFF85,(byte)0x14,(byte)0x0,
        (byte)0x3B};

  public void serialize(Document doc, XSQLPageRequest env) throws Throwable {
    env.setContentType("image/gif");
    OutputStream os = env.getOutputStream();
    os.write(okGif, 0, okGif.length);
    os.flush();
  }
}
```
Using the XSQL command-line utility, the binary information is written to the target output file. Using the `XSQLRequest` API, two constructors exist that allow the caller to supply the target `OutputStream` to use for the results of page processing.

Note that your serializer must either call `getWriter()` for textual output or `getOutputStream()` for binary output but not both. Calling both in the same request raises an error.

Using a Custom XSQL Connection Manager for JDBC Datasources

As an alternative to defining your named connections in the XSQL configuration file, you can use one of the two provided `XSQLConnectionManager` implementations. These implementations enable you to use your servlet container's JDBC Datasource implementation and related connection pooling features.

This XSQL pages framework provides the following alternative connection manager implementations:

- `oracle.xml.xsql.XSQLDatasourceConnectionManager`

  Consider using this connection manager if your servlet container's datasource implementation does not use the Oracle JDBC driver. Features of the XSQL pages system such as `<xsql:ref-cursor-function>` and `<xsql:include-owa>` are not available when you do not use an Oracle JDBC driver.

- `oracle.xml.xsql.XSQLOracleDatasourceConnectionManager`

  Consider using this connection manager when your datasource implementation returns JDBC `PreparedStatement` and `CallableStatement` objects that implement the `oracle.jdbc.PreparedStatement` and `oracle.jdbc.CallableStatement` interfaces. The Oracle Application Server has a datasource implementation that performs this task.

When using either of the preceding alternative connection manager implementations, the value of the connection attribute in your XSQL page template is the JNDI name used to look up your desired datasource. For example, the value of the connection attribute might look like the following:

- `jdbc/scottDS`
- `java:comp/env/jdbc/MyDatasource`

If you are not using the default XSQL pages connection manager, then needed connection pooling functionality must be provided by the alternative connection manager implementation. In the case of the preceding two options based on JDBC datasources, you must properly configure your servlet container to supply the connection pooling. See your servlet container documentation for instructions on how to properly configure the datasources to offer pooled connections.

Writing Custom XSQL Connection Managers

You can provide a custom connection manager to replace the built-in connection management mechanism. To provide a custom connection manager implementation, you must perform the following steps:

1. Write a connection manager factory class that implements the `oracle.xml.xsql.XSQLConnectionManagerFactory` interface.
Performing XSQL Customizations

2. Write a connection manager class that implements the oracle.xml.xsql.XSQLConnectionManager interface.

3. Change the name of the XSQLConnectionManagerFactory class in your XSQL configuration file.

The XSQL servlet uses your connection management scheme instead of the XSQL pages default scheme.

You can set your custom connection manager factory as the default connection manager factory by providing the class name in the XSQL configuration file. Set the factory in the following section:

```xml
<!--
| Set the name of the XSQL Connection Manager Factory implementation. The class must implement the oracle.xml.xsql.XSQLConnectionManagerFactory interface.
| If unset, the default is to use the built-in connection manager implementation in oracle.xml.xsql.XSQLConnectionManagerFactoryImpl
+-->
<connection-manager>
    <factory>oracle.xml.xsql.XSQLConnectionManagerFactoryImpl</factory>
</connection-manager>
```

In addition to specifying the default connection manager factory, you can associate a custom connection factory with a XSQLRequest object by using APIs provided.

The responsibility of the XSQLConnectionManagerFactory is to return an instance of an XSQLConnectionManager for use by the current request. In a multithreaded environment such as a servlet engine, the XSQLConnectionManager object must ensure that a single XSQLConnection instance is not used by two different threads. This aim is realized by marking the connection as in use for the time between the getConnection() and releaseConnection() method calls. The default XSQL connection manager implementation automatically pools named connections and adheres to this thread-safe policy.

If your custom implementation of XSQLConnectionManager implements the optional oracle.xml.xsql.XSQLConnectionManagerCleanup interface, then your connection manager can clean up any resources it has allocated. For example, if your servlet container invokes the destroy() method on the XSQLServlet servlet, which can occur during online administration of the servlet for example, the connection manager has a chance to clean up resources as part of the servlet destruction process.

Accessing Authentication Information in a Custom Connection Manager

To use the HTTP authentication mechanism to get the username and password to connect to the database, write a customized connection manager. You can then invoke a getConnection() method to obtain the needed information.

You can write a Java program that follows these steps:

1. Pass an instance of the oracle.xml.xsql.XSQLPageRequest interface to the getConnection() method.
2. Invoke getRequestType() to ensure that the request type is Servlet.
3. Cast the XSQLPageRequest object to an XSQLServletPageRequest.
4. Call getHttpServletRequest() on the result of the preceding step.
5. Obtain the authentication information from the javax.servlet.http.HttpServletRequest object returned by the previous call.
Implementing a Custom XSQLErrorHandler

You may want to control how serious page processor errors such as an unavailable connection are reported to users. You can achieve this task by implementing the oracle.xml.xsql.XSQLErrorHandler interface. The interface contains the following single method signature:

```java
public interface XSQLErrorHandler {
    public void handleError(XSQLError err, XSQLPageRequest env);
}
```

You can provide a class that implements the XSQLErrorHandler interface to customize how the XSQL pages processor writes error messages. The new XSQLError object encapsulates the error information and provides access to the error code, formatted error message, and so on.

**Example 15–24** illustrates a sample implementation of XSQLErrorHandler.

**Example 15–24  myErrorHandler class**

```java
package example;
import oracle.xml.xsql.*;
import java.io.*;
public class myErrorHandler implements XSQLErrorHandler {
    public void logError(XSQLError err, XSQLPageRequest env) {
        // Must set the content type before writing anything out
        env.setContentType("text/html");
        PrintWriter pw = env.getErrorWriter();
        pw.println(<H1>ERROR</H1><hr>" + err.getMessage());
    }
}
```

You can control which custom XSQLErrorHandler implementation is used in the following distinct ways:

1. Define the name of a custom XSQLErrorHandler implementation class in the XSQL configuration file. You must provide the fully-qualified class name of your error handler class as the value of the /XSQLConfig/processor/error-handler/class entry.

   If the XSQL processor can load this class, and if it correctly implements the XSQLErrorHandler interface, then it uses this class as a singleton and replaces the default implementation globally wherever page processor errors are reported.

2. Override the error writer on a per page basis by using the errorHandler (or xsql:$errorHandler) attribute on the document element of the page. The attribute value is the fully-qualified class name of a class that implements the XSQLErrorHandler interface. This class reports the errors for this page only. The class is instantiated on each page request by the page engine.

You can use a combination of the preceding approaches if needed.

Providing a Custom XSQL Logger Implementation

You can optionally register custom code to handle the logging of the start and end of each XSQL page request. Your custom logger code must provide an implementation of the oracle.xml.xsql.XSQLLoggerFactory and oracle.xml.xsql.XSQLLogger interfaces.

The XSQLLoggerFactory interface contains the following single method:

```java
public interface XSQLLoggerFactory {
```

public XSQLLogger create(XSQLPageRequest env);
}

You can provide a class that implements the XSQLLoggerFactory interface to decide how XSQLLogger objects are created (or reused) for logging. The XSQL processor holds a reference to the XSQLLogger object returned by the factory for the duration of a page request. The processor uses it to log the start and end of each page request by invoking the logRequestStart() and logRequestEnd() methods.

The XSQLLogger interface is as follows:
public interface XSQLLogger {
    public void logRequestStart(XSQLPageRequest env);
    public void logRequestEnd(XSQLPageRequest env);
}

The classes in Example 15–25 and Example 15–26 illustrate a trivial implementation of a custom logger. The XSQLLogger implementation in Example 15–25 notes the time the page request started. It then logs the page request end by printing the name of the page request and the elapsed time to System.out.

Example 15–25 SampleCustomLogger Class
package example;
import oracle.xml.xsql.*;
public class SampleCustomLogger implements XSQLLogger {
    long start = 0;
    public void logRequestStart(XSQLPageRequest env) {
        start = System.currentTimeMillis();
    }
    public void logRequestEnd(XSQLPageRequest env) {
        long secs = System.currentTimeMillis() - start;
        System.out.println("Request for " + env.getSourceDocumentURI() + " took "+ secs + "ms");
    }
}

The factory implementation is shown in Example 15–26.

Example 15–26 SampleCustomLoggerFactory Class
package example;
import oracle.xml.xsql.*;
public class SampleCustomLoggerFactory implements XSQLLoggerFactory {
    public XSQLLogger create(XSQLPageRequest env) {
        return new SampleCustomLogger();
    }
}

To register a custom logger factory, edit the XSQLConfig.xml file and provide the name of your custom logger factory class as the content to the /XSQLConfig/processor/logger/factory element. Example 15–27 illustrates this technique.

Example 15–27 Registering a Custom Logger Factory
<XSQLConfig>
    ...
    <processor>
        ...

By default, `<logger>` section is commented out. There is no default logger.
This part contains chapters describing how the Oracle XDK is used for development in C.

This part contains the following chapters:

- Chapter 16, "Getting Started with C XDK Components"
- Chapter 17, "Using the XSLT and XVM Processors for C"
- Chapter 18, "Using the XML Parser for C"
- Chapter 19, "Using Binary XML for C"
- Chapter 20, "Using the XML Schema Processor for C"
- Chapter 21, "Determining XML Differences Using C"
- Chapter 22, "Using SOAP with the C XDK"
The XDK C components are the building blocks for reading, manipulating, transforming, and validating XML.

This chapter contains the following topics:

- Installing C XDK Components
- Configuring the UNIX Environment for C XDK Components
- Configuring the Windows Environment for C XDK Components
- Overview of the Unified C API
- Globalization Support for the C XDK Components

**Installing C XDK Components**

The C XDK components are included with Oracle Database. This chapter assumes that you have installed XDK with Oracle Database and also installed the demo programs on the Oracle Database Examples media. Refer to "Installing the XDK" on page 1-17 for installation instructions and a description of the XDK directory structure.

Example 16–1 shows the UNIX directory structure for the XDK demos and the libraries used by the XDK components. The $ORACLE_HOME/xdk/demo/c subdirectories contain sample programs and data files for the XDK for C components. The chapters in Part II, "XDK for C" explain how to understand and use these programs.

**Example 16–1  C XDK Libraries, Header Files, Utilities, and Demos**

```
- Oracle_home_directory
  | - bin/
  |    | schema
  |    | xml
  |    | xmlcg
  |    | xsl
  |    | xvm
  | - lib/
  |    | libcore11.a
  |    | libcoresh11.so
  |    | libnls11.a
  |    | libunls11.a
  |    | libxml11.a
  |    | libxmlsh10.a
  | - xdk/
  |    | demo/
  |    |    | c/
  |    |    |    | dom/
```
The subdirectories contain sample programs and data files for the C XDK components. The chapters in Part II, "XDK for C" explain how to use these programs to gain an understanding of the most important C features.

See Also: "Overview of Oracle XML Developer's Kit (XDK)" on page 1-1 for a list of the XDK C components

Configuring the UNIX Environment for C XDK Components

This section contains the following topics:

■ C XDK Component Dependencies on UNIX
■ Setting C XDK Environment Variables on UNIX
■ Testing the C XDK Runtime Environment on UNIX
■ Setting Up and Testing the C XDK Compile-Time Environment on UNIX
■ Verifying the C XDK Component Version on UNIX

C XDK Component Dependencies on UNIX

The C libraries described in this section are located in $ORACLE_HOME/lib. The XDK C and C++ components are contained in the following library:

libxml11.a

The following XDK components are contained in the library:

■ XML parser, which checks an XML document for well-formedness, optionally validates it against a DTD or XML Schema, and supports DOM and SAX interfaces for programmatic access
■ XSLT processor, which transforms an XML document into another XML document
■ XSLT compiler, which compiles XSLT stylesheets into byte code for use by the XSLT Virtual Machine
■ XSLTVM, which is an XSLT transformation engine
■ XML Schema processor, which validates XML files against an XML schema
Table 16–1 describes the Oracle CORE and Globalization Support libraries on which the XDK C components (UNIX) depend.

**Table 16–1 Dependent Libraries of XDK C Components on UNIX**

<table>
<thead>
<tr>
<th>Component Library</th>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE library</td>
<td>libcore11.a</td>
<td>Contains the C runtime functions that enable portability across platforms.</td>
</tr>
<tr>
<td>CORE Dynamic linking library</td>
<td>libcoresh11.so</td>
<td>C runtime library that supports dynamic linking on UNIX platforms.</td>
</tr>
<tr>
<td>Globalization Support common library</td>
<td>libnls11.a</td>
<td>Supports the UTF-8, UTF-16, and ISO-8859-1 character sets. This library depends on the environment to locate encoding and message files.</td>
</tr>
<tr>
<td>Globalization Support library for Unicode</td>
<td>libunls11.a</td>
<td>Supports the character sets described in Oracle Database Globalization Support Guide. This library depends on the environment to locate encoding and message files.</td>
</tr>
</tbody>
</table>

**Setting C XDK Environment Variables on UNIX**

Table 16–2 describes the UNIX environment variables required for use with the XDK C components.

**Table 16–2 UNIX Environment Settings for XDK C Components**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ORA_NLS10</td>
<td>Sets the location of the Globalization Support character-encoding definition files. The encoding files represent a subset of character sets available in Oracle Database.</td>
<td>Set to the location of the Globalization Support data files. Set the variable as follows: setenv $ORA_NLS10 $ORACLE_HOME/nls/data</td>
</tr>
<tr>
<td>$ORA_XML_MESG</td>
<td>Sets the location of the XML error message files. Files ending in .msb are machine-readable and required at runtime. Files ending in .msg are human-readable and contain cause and action descriptions for each error.</td>
<td>Set to the path of the mesg directory. For example: setenv $ORA_XML_MESG $ORACLE_HOME/xdk/mesg</td>
</tr>
<tr>
<td>$PATH</td>
<td>Sets the location of the C XDK executables.</td>
<td>You can set the PATH as follows: setenv PATH $(PATH):${ORACLE_HOME}/bin</td>
</tr>
</tbody>
</table>

**Testing the C XDK Runtime Environment on UNIX**

You can test your UNIX runtime environment by running any of the utilities described in Table 16–3.
Run these utilities with no options to display the usage help. Run the utilities with the -hh flag for complete usage information.

### Setting Up and Testing the C XDK Compile-Time Environment on UNIX

Table 16–4 describes the header files required for compilation of the C components. These files are located in $ORACLE_HOME/xdk/include. Note that your runtime environment must be set up before you can compile your code.

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oratypes.h</td>
<td>Includes the private Oracle C datatypes.</td>
</tr>
<tr>
<td>oraxml.h</td>
<td>Includes the Oracle9i XML ORA datatypes and the public ORA APIs included in libxml.a (for backward compatibility only). Use xml.h instead.</td>
</tr>
<tr>
<td>oraxmlcg.h</td>
<td>Includes the C APIs for the C++ class generator (for backward compatibility only).</td>
</tr>
<tr>
<td>oraxsd.h</td>
<td>Includes the Oracle9i XSD validator datatypes and APIs (for backward compatibility only).</td>
</tr>
<tr>
<td>xml.h</td>
<td>Handles the unified DOM APIs transparently, whether you use them through OCI or standalone. It replaces oraxml.h, which is deprecated.</td>
</tr>
<tr>
<td>xmlerr.h</td>
<td>Includes the XML errors and their numbers.</td>
</tr>
<tr>
<td>xmlotn.h</td>
<td>Includes the other headers depending on whether you compile standalone or use OCI.</td>
</tr>
<tr>
<td>xmlproc.h</td>
<td>Includes the Oracle XML datatypes and XML public parser APIs in libxml11.a.</td>
</tr>
<tr>
<td>xmlsch.h</td>
<td>Includes the Oracle XSD validator public APIs.</td>
</tr>
<tr>
<td>xmlptr.h</td>
<td>Includes the XPointer datatypes and APIs, which are not currently documented or supported.</td>
</tr>
<tr>
<td>xmlxsl.h</td>
<td>Includes the XSLT processor datatypes and public APIs.</td>
</tr>
<tr>
<td>xmlxvm.h</td>
<td>Includes the XSLT compiler and VM datatypes and public APIs.</td>
</tr>
</tbody>
</table>
Testing the C XDK Compile-Time Environment on UNIX
The simplest way to test your compile-time environment is to run the `make` utility on the sample programs, which are located on the Examples media rather than on the Oracle Database CD. After you install the demos, they will be located in `$ORACLE_HOME/xdk/demo/c`. A README in the same directory provides compilation instructions and usage notes.

Build and run the sample programs by executing the following commands at the system prompt:

```
cd $ORACLE_HOME/xdk/demo/c
make
```

Verifying the C XDK Component Version on UNIX
To obtain the version of XDK you are working with, change directory into `$ORACLE_HOME/lib` and run the following command:

```
strings libxml11.a | grep -i developers
```

Configuring the Windows Environment for C XDK Components
This section contains the following topics:

- C XDK Component Dependencies on Windows
- Setting C XDK Environment Variables on Windows
- Testing the C XDK Runtime Environment on Windows
- Setting Up and Testing the C XDK Compile-Time Environment on Windows
- Using the C XDK Components with Visual C/C++ on Windows

C XDK Component Dependencies on Windows
The C libraries described in this section are located in `%ORACLE_HOME%\lib`. The XDK C components are contained in the following library:

```
libxml11.dll
```

The following XDK components are contained in the library:

- XML parser
- XSLT processor
- XSLT compiler
- XSLT VM
- XML Schema processor

Table 16–5 describes the Oracle CORE and Globalization Support libraries on which the XDK C components (Windows) depend.

<table>
<thead>
<tr>
<th>Component</th>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE library</td>
<td>libcore11.dll</td>
<td>Contains the runtime functions that enable portability across platforms.</td>
</tr>
</tbody>
</table>
### Setting C XDK Environment Variables on Windows

Table 16–6 describes the Windows environment variables required for use with the XDK C components.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Setting</th>
</tr>
</thead>
</table>
| %ORA_NLS10%      | Sets the location of the Globalization Support character-encoding definition files. The encoding files represent a subset of character sets available in Oracle Database. | This variable should be set to the location of the Globalization Support data files. Set the variable as follows: set ORA_NLS10=%ORACLE_HOME%
ls\data |
| %ORA_XML_MESG%   | Sets the location of the XML error message files. Files ending in .msb are machine-readable and required at runtime. Files ending in .msg are human-readable and contain cause and action descriptions for each error. | Set to the path of the mesg directory. For example: set ORA_XML_MESG=%ORACLE_HOME%\xdk\mesg |
| %PATH%           | Sets the location of the C XDK DLLs and executables.                         | You can set the PATH as follows: path %path%;%ORACLE_HOME%\bin           |

### Testing the C XDK Runtime Environment on Windows

You can test your Windows runtime environment by running any of the utilities described in Table 16–7.

<table>
<thead>
<tr>
<th>Executable</th>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema.exe</td>
<td>%ORACLE_HOME%\bin</td>
<td>C XML Schema validator</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>See Also:</strong> &quot;Using the C XML Schema Processor Command-Line Utility&quot; on page 20-2</td>
</tr>
<tr>
<td>xml.exe</td>
<td>%ORACLE_HOME%\bin</td>
<td>C XML parser</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>See Also:</strong> &quot;Using the C XML Parser Command-Line Utility&quot; on page 18-9</td>
</tr>
</tbody>
</table>
Configuring the Windows Environment for C XDK Components

Table 16–7 (Cont.) C/C++ XDK Utilities on Windows
Executable

Directory

Description

xmlcg.exe

%ORACLE_HOME%\bin

C++ class generator
See Also: "Using the XML C++ Class Generator Command-Line Utility"
on page 29-2

xsl.exe

%ORACLE_HOME%\bin

C XSLT processor
See Also: "Using the C XSLT Processor Command-Line Utility" on
page 17-4

xvm.exe

%ORACLE_HOME%\bin

C XVM processor
See Also: "Using the XVM Processor Command-Line Utility" on
page 17-3

Run these utilities with no options to display the usage help. Run the utilities with the
-hh flag for complete usage information.

Setting Up and Testing the C XDK Compile-Time Environment on Windows
Table 16–4 in the section "Setting Up and Testing the C XDK Compile-Time
Environment on UNIX" on page 16-4 describes the header files required for
compilation of the C components on Windows. The relative filenames are the same on
both UNIX and Windows installations.
On Windows the header files are located in %ORACLE_HOME%\xdk\include. Note that
you must set up your runtime environment before you can compile your code.

Testing the C XDK Compile-Time Environment on Windows
You can test your compile-time environment by compiling the demo programs, which
are located in %ORACLE_HOME%\xdk\demo\c after you install them from the Oracle
Database Examples media. A README in the same directory provides compilation
instructions and usage notes. Before you compile the demo programs, edit the
Make.bat files as described in "Editing the Make.bat Files on Windows" on page 16-7.
Editing the Make.bat Files on Windows Each subfolder of the %ORACLE_HOME%\xdk\demo\c
folder contains a Make.bat file. Update the Make.bat file in each folder by adding the
path of the libraries and the header files to the compile command. You should not
need to edit the paths in the :LINK section because /libpath:%ORACLE_HOME%\lib
already points to the C libraries. The section of a Make.bat file in Example 16–2 uses
bold text to show the path that you need to include.
Example 16–2

Editing a C XDK Make.bat File on Windows

:COMPILE
set filename=%1
cl -c -Fo%filename%.obj %opt_flg% /DCRTAPI1=_cdecl /DCRTAPI2=_cdecl /nologo /Zl
/Gy /DWIN32 /D_WIN32 /DWIN_NT /DWIN32COMMON /D_DLL /D_MT /D_X86_=1
/Doratext=OraText -I. -I..\..\..\include -I%ORACLE_HOME%\xdk\include %filename%.c
goto :EOF
:LINK
set filename=%1
link %link_dbg% /out:..\..\..\..\bin\%filename%.exe
/libpath:%ORACLE_HOME%\lib /libpath:..\..\..\..\lib
%filename%.obj oraxml10.lib user32.lib kernel32.lib msvcrt.lib ADVAPI32.lib
oldnames.lib winmm.lib

Getting Started with C XDK Components 16-7


Setting the C XDK Compiler Path on Windows

The demo `make.bat` file assumes that you are using the `cl.exe` compiler, which is freely available with the Microsoft .NET Framework Software Development Kit (SDK).

To set the path for the `cl.exe` compiler on Windows XP, follow these steps:

1. In the Start menu, select Settings and then Control Panel.
2. Double-click System.
3. In the System Properties dialogue box, select the Advanced tab and click Environment Variables.
4. In System variables, select Path and click Edit.
5. Append the path of `cl.exe` to the `%PATH%` variable and click OK.

Build and run the sample programs by executing the following commands at the system prompt:

```
cd $ORACLE_HOME/xdk/demo/c
make
```

Using the C XDK Components with Visual C/C++ on Windows

You can set up a project in Microsoft Visual C/C++ and use it for the demos included in the XDK.

Setting a Path for a Project in Visual C/C++ on Windows

Follow these steps to set the path for a project:

1. Open a workspace in Visual C++ and include the *.c files for your project.
2. Navigate to the Tools menu and select Options.
3. Select the Directories tab and set your include path to `%ORACLE_HOME%/xdk/include` as shown in the example in Figure 16–1.

Figure 16–1 Setting the Include Path in Visual C/C++
Setting the Library Path in Visual C/C++ on Windows

Follow these steps to set the library path for a project:

1. Open a workspace in Visual C++ and include the *.c files for your project.
2. Navigate to the Tools menu and select Options.
3. Select the Directories tab and set your library path to `%ORACLE_HOME%/lib` as shown in the example in Figure 16–2.

Figure 16–2  Setting the Static Library Path in Visual C/C++

4. After setting the paths for the static libraries in `%ORACLE_HOME%/lib`, set the library name in the compiling environment of Visual C++. Navigate to the Project menu in the menu bar and select Settings.
5. Select the Link tab in the Object/Library Modules field and enter the names of XDK C components libraries, as shown in the example in Figure 16–3.
Overview of the Unified C API

The unified C API is a programming interface that unifies the functionality required by both the XDK and Oracle XML DB. This API is used primarily by XSLT and XML Schema.

As shown in Table 16–4, the unified C API is declared in the xml.h header file. Table 16–8 summarizes the C XDK APIs. Refer to Oracle XML API Reference for complete documentation.

Table 16–8 Summary of the XDK C APIs

<table>
<thead>
<tr>
<th>Package</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callback APIs</td>
<td>Define macros that declare functions (or function pointers) for XML callbacks.</td>
</tr>
<tr>
<td>DOM APIs</td>
<td>Parse and manipulate XML documents with DOM. The API follows the DOM 2.0 standard as closely as possible, although it changes some names when mapping from the objected-oriented DOM specification to the flat C namespace. For example, the overloaded getName() methods become getAttrName().</td>
</tr>
<tr>
<td>Range APIs</td>
<td>Create and manipulate Range objects.</td>
</tr>
<tr>
<td>SAX APIs</td>
<td>Enable event-based XML parsing with SAX.</td>
</tr>
<tr>
<td>Schema APIs</td>
<td>Assemble multiple XML schema documents into a single schema that can be used to validate a specific instance document.</td>
</tr>
<tr>
<td>Traversal APIs</td>
<td>Enable document traversal and navigation of DOM trees.</td>
</tr>
<tr>
<td>XML APIs</td>
<td>Define an XML processor in terms of how it must read XML data and the information it must provide to the application.</td>
</tr>
</tbody>
</table>
The API accomplishes the unification of the functions by conforming contexts. A top-level XML context (`xmlctx`) shares common information between cooperating XML components. This context defines information about the following:

- Data encoding
- Error message language
- Low-level allocation callbacks

An application needs this information before it can parse a document and provide programmatic access through DOM or SAX interfaces.

Both XDK and Oracle XML DB require different startup and tear-down functions for the top-level and service contexts. The initialization function takes implementation-specific arguments and returns a conforming context.

The unification is made possible by using conforming contexts. A conforming context means that the returned context must begin with a `xmlctx`; it may have any additional implementation-specific parts after the standard header.

After an application obtains `xmlctx`, it uses unified DOM calls, all of which take an `xmlctx` as the first argument.

### Globalization Support for the C XDK Components

The C XDK parser supports over 300 IANA character sets. These character sets include those listed in "Character Sets Supported by the XDK for C" on page 31-6. Note the following considerations when working with character sets:

- It is recommended that you use IANA character set names for interoperability with other XML parsers.
- XML parsers are only required to support UTF-8 and UTF-16, so these character sets are preferable.
- The default input encoding ("incoding") is UTF-8. If an input document's encoding is not self-evident (by HTTP character set, Byte Order Mark, XMLDecl, and so on), then the default input encoding is assumed. It is recommended that you set the default encoding explicitly if using only single byte character sets such as US-ASCII or any of the ISO-8859 character sets because single-byte performance is fastest. The flag `XML_FLAG_FORCE_INCODING` specifies that the default input encoding should always be applied to input documents, ignoring any BOM or XMLDecl. Nevertheless, a protocol declaration such as HTTP character set is always honored.
- Choose the data encoding for DOM and SAX ("outcoding") carefully. Single-byte encodings are the fastest, but can represent only a very limited set of characters. Next fastest is Unicode (UTF-16), and slowest are the multibyte encodings such as UTF-8. If input data cannot be converted to the outcoding without loss, then an error occurs. For maximum utility, you should use a Unicode-based outcoding.

<table>
<thead>
<tr>
<th>Package</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPath APIs</td>
<td>Process XPath-related types and interfaces.</td>
</tr>
<tr>
<td>XPointer APIs</td>
<td>Locate nodes in an XML document.</td>
</tr>
<tr>
<td>XSLT APIs</td>
<td>Perform XSL processing.</td>
</tr>
<tr>
<td>XSLTVM APIs</td>
<td>Implement a virtual machine that can run compiled XSLT code.</td>
</tr>
</tbody>
</table>

**Table 16–8 (Cont.) Summary of the XDK C APIs**
because Unicode can represent any character. If outcoding is not specified, then it defaults to the incoding of the first document parsed.
This chapter contains these topics:

- XVM Processor
- XSLT Processor
- Using the Demo Files Included with the Software

---

**Note:** Use the new unified C API for new XDK and Oracle XML DB applications. The old C functions are deprecated and supported only for backward compatibility, but will not be enhanced. They will be removed in a future release.

The new C API is described in Chapter 18, "Using the XML Parser for C".

---

**XVM Processor**

The Oracle XVM Package implements the XSL Transformation (XSLT) language as specified in the W3C Recommendation of 16 November 1999. The package includes XSLT Compiler and XSLT Virtual Machine (XVM). The implementation by Oracle of the XSLT compiler and the XVM enables compilation of XSLT (Version 1.0) into bytecode format, which is executed by the virtual machine. XSLT Virtual Machine is the software implementation of a "CPU" designed to run compiled XSLT code. The virtual machine assumes a compiler compiling XSLT stylesheets to a sequence of bytecodes or machine instructions for the "XSLT CPU". The bytecode program is a platform-independent sequence of 2-byte units. It can be stored, cached and run on different XVMs. The XVM uses the bytecode programs to transform instance XML documents. This approach clearly separates compile-time from run-time computations and specifies a uniform way of exchanging data between instructions. The benefits of this approach are:

- An XSLT stylesheet can be compiled, saved in a file, and re-used often, even on different platforms.
- The XVM is significantly faster and uses less memory than other XSLT processors.
- The bytecodes are not language-dependent. There is no difference between code generated from a C or C++ XSLT compiler.

**XVM Usage Example**

A typical scenario of using the package APIs has the following steps:

1. Create and use an XML meta-context object.
xctx = XmlCreate(&err,...);

2. Create and use an XSLT compiler object.
   comp = XmlXvmCreateComp(xctx);

3. Compile an XSLT stylesheet or XPath expression and store or cache the resulting bytecode.
   code = XmlXvmCompileFile(comp, xslFile, baseuri, flags, &err);
   
   or
   code = XmlXvmCompileDom (comp, xslDomdoc, flags, &err);
   
   or
   code = XmlXvmCompileXPath (comp, xpathexp, namespaces, &err);

4. Create and use an XVM object. The explicit stack size setting is needed when XVM terminates with a "Stack Overflow" message or when smaller memory footprints are required. See XmlXvmCreate().
   vm = XmlXvmCreate(xctx, "StringStack", 32, "NodeStack", 24, NULL);

5. Set the output (optional). Default is a stream.
   err = XmlXvmSetOutputDom (vm, NULL);
   
   or
   err = XmlXvmSetOutputStream(vm, &xvm_stream);
   
   or
   err = XmlXvmSetOutputSax(vm, &xvm_callback, NULL);

6. Set a stylesheet bytecode to the XVM object. Can be repeated with other bytecode.
   len = XmlXvmGetBytecodeLength(code, &err);
   err = XmlXvmSetBytecodeBuffer(vm, code, len);
   
   or
   err = XmlXvmSetBytecodeFile (vm, xslBytecodeFile);

7. Transform an instance XML document or evaluate a compiled XPath expression. Can be repeated with the same or other XML documents.
   err = XmlXvmTransformFile(vm, xmlFile, baseuri);
   
   or
   err = XmlXvmTransformDom (vm, xmlDomdoc);
   
   or
   obj  = (xvmobj*)XmlXvmEvaluateXPath (vm, code, 1, 1, node);

8. Get the output tree fragment (if DOM output is set at step 5).
   node = XmlXvmGetOutputDom (vm);

9. Delete the objects.
Using the XVM Processor Command-Line Utility

The XVM processor is accessed from the command-line this way:

```
xvm
```

Usage:

```
xvm options xslfile xmlfile
xvm options xpath xmlfile
```

Options:

```
-c  Compile xslfile. The bytecode is in "xmlfile.xvm".
-ct Compile xslfile and transform xmlfile.
-t Transform xmlfile using bytecode from xslfile.
-xc Compile xpath. The bytecode is in "code.xvm".
-xct Compile and evaluate xpath with xmlfile.
-xt Evaluate XPath bytecode from xpath with xmlfile.
```

Examples:

```
xvm -ct  db.xsl  db.xml
xvm -t   db.xvm db.xml
xvm -xct "doc/employee[15]/family" db.xml
```

Accessing XVM Processor for C

Oracle XVM Processor for C is part of the standard installation of Oracle Database.

See Also:

- Oracle Database XML C API Reference "XSLTVM APIs for C"
- http://www.oracle.com/technology/tech/xml/

XSLT Processor

The Oracle XSL/XPath Package implements the XSL Transformation (XSLT) language as specified in the W3C Recommendation of 16 November 1999. The package includes the XSLT processor and XPath Processor. The Oracle implementation of the XSLT processor follows the more common design approach, which melts 'compiler' and 'processor' into one object.

XSLT Processor Usage Example

A typical scenario of using the package APIs has the following steps:

1. Create and use an XML meta-context object.

   ```c
   xctx = XmlCreate(&err,...);
   ```

2. Parse the XSLT stylesheet.

   ```c
   xslDomdoc = XmlLoadDom(xctx, &err, "file", xslFile, "base_uri", baseuri, NULL);
   ```

3. Create an XSLT processor for the stylesheet
XSLT Processor

xslproc = XmlXslCreate(xctx, xslDomdoc, baseuri, &err);

4. Parse the instance XML document.
   xmlDomdoc = XmlLoadDom(xctx, &err, "file", xmlFile, "base_uri", baseuri, NULL);

5. Set the output (optional). Default is DOM.
   err = XmlXslSetOutputStream(xslproc, &stream);

6. Transform the XML document. This step can be repeated with the same or other XML documents.
   err = XmlXslProcess(xslproc, xmlDomdoc, FALSE);

7. Get the output (if DOM).
   node = XmlXslGetOutput(xslproc);

8. Delete objects.
   XmlXslDestroy(xslproc);
   XmlDestroy(xctx);

XPath Processor Usage Example

A typical scenario of using the package APIs has the following steps:

1. Create and use an XML meta-context object.
   xctx = XmlCreate(&err, ...);

2. Parse the XML document or get the current node from already existing DOM.
   node = XmlLoadDom(xctx, &err, "file", xmlFile, "base_uri", baseuri, NULL);

3. Create an XPath processor.
   xptproc = XmlXPathCreateCtx(xctx, NULL, node, 0, NULL);

4. Parse the XPath expression.
   exp = XmlXPathParse(xptproc, xpathexpr, &err);

5. Evaluate the XPath expression.
   obj = XmlXPathEval(xptproc, exp, &err);

6. Delete the objects.
   XmlXPathDestroyCtx(xptproc);
   XmlDestroy(xctx);

Using the C XSLT Processor Command-Line Utility

You can call the C Oracle XSLT processor as an executable by invoking bin/xsl:

xsl [switches] stylesheet instance
or
xsl -f [switches] [document filespec]

If no style sheet is provided, no output is generated. If there is a style sheet, but no output file, output goes to stdout.
Table 17–1 lists the command line options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-B BaseUri</td>
<td>Set the Base URI for XSLT processor: BaseUri of <a href="http://pqr/xsl.txt">http://pqr/xsl.txt</a> resolves pqr.txt to <a href="http://pqr/pqr.txt">http://pqr/pqr.txt</a></td>
</tr>
<tr>
<td>-e encoding</td>
<td>Specify default input file encoding (-ee to force).</td>
</tr>
<tr>
<td>-E encoding</td>
<td>Specify DOM or SAX encoding.</td>
</tr>
<tr>
<td>-f</td>
<td>File - interpret as filespec, not URI.</td>
</tr>
<tr>
<td>-G xptrexprs</td>
<td>Evaluates XPointer schema examples given in a file.</td>
</tr>
<tr>
<td>-h</td>
<td>Help - show this usage. (Use -hh for more options.)</td>
</tr>
<tr>
<td>-hh</td>
<td>Show complete options list.</td>
</tr>
<tr>
<td>-i n</td>
<td>Number of times to iterate the XSLT processing.</td>
</tr>
<tr>
<td>-l language</td>
<td>Language for error reporting.</td>
</tr>
<tr>
<td>-o XSLoutfile</td>
<td>Specifies output file of XSLT processor.</td>
</tr>
<tr>
<td>-v</td>
<td>Version - display parser version then exit.</td>
</tr>
<tr>
<td>-V var value</td>
<td>Test top-level variables in C XSLT.</td>
</tr>
<tr>
<td>-w</td>
<td>Whitespace - preserve all whitespace.</td>
</tr>
<tr>
<td>-W</td>
<td>Warning - stop parsing after a warning.</td>
</tr>
</tbody>
</table>

Accessing Oracle XSLT processor for C

Oracle XSLT processor for C is part of the standard installation of Oracle Database.

See Also:
- Oracle Database XML C API Reference "XSLT APIs for C"
- Oracle Database XML C API Reference "XPath APIs for C"

Using the Demo Files Included with the Software

$ORACLE_HOME/xdk/demo/c/parser/ directory contains several XML applications to illustrate how to use the XSLT for C.

Table 17–2 lists the files in that directory:

<table>
<thead>
<tr>
<th>Sample File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLSample.c</td>
<td>Source for XSLSample program</td>
</tr>
<tr>
<td>XSLSample.std</td>
<td>Expected output from XSLSample</td>
</tr>
<tr>
<td>class.xml</td>
<td>XML file that can be used with XSLSample</td>
</tr>
<tr>
<td>iden.xsl</td>
<td>Stylesheet that can be used with XSLSample</td>
</tr>
<tr>
<td>cleo.xml</td>
<td>XML version of Shakespeare’s play</td>
</tr>
<tr>
<td>XVMSample.c</td>
<td>Sample usage of XSLT Virtual Machine and compiler. It takes two filenames as input - XML file and XSLT stylesheet file.</td>
</tr>
</tbody>
</table>
Building the C Demo Programs for XSLT

Change directories to the demo directory and read the README file. This will explain how to build the sample programs according to your operating system.

Here is the usage of XSLT processor sample XSLSample, which takes two files as input, the XML file and the XSLT stylesheet:

XSLSample xmlfile xslss

<table>
<thead>
<tr>
<th>Sample File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XVMXPathSample.c</td>
<td>Sample usage of XSLT Virtual Machine and compiler. It takes XML file name and XPath expression as input. Generates the result of the evaluated XPath expression.</td>
</tr>
<tr>
<td>XSLXPathSample.c</td>
<td>Sample usage of XSL/XPath processor. It takes XML file name and XPath expression as input. Generates the result of the evaluated XPath expression.</td>
</tr>
</tbody>
</table>

Table 17–2 (Cont.) XSLT for C Demo Files
This chapter contains these topics:

- Introduction to the XML Parser for C
- Using the XML Parser for C
- Using the DOM API for C
- Using the SAX API for C
- What is the XML Pull Parser for C?
- Using OCI and the XDK C API

Introduction to the XML Parser for C

This section contains the following topics:

- Prerequisites
- Standards and Specifications

See Also: "Introduction to the XML Parsing for Java" on page 4-1 for a generic introduction to XML parsing with DOM and SAX. Much of the information in the introduction is language-independent and applies equally to C.

Prerequisites

The Oracle XML parser for C reads an XML document and uses DOM or SAX APIs to provide programmatic access to its content and structure. You can use the parser in validating or nonvalidating mode. A pull parser is also available.

This chapter assumes that you are familiar with the following technologies:

- **Document Object Model (DOM)**. DOM is an in-memory tree representation of the structure of an XML document.
- **Simple API for XML (SAX)**. SAX is a standard for event-based XML parsing.
- **What is the XML Pull Parser for C?**. Pull Parser uses XML events.
- **Document Type Definition (DTD)**. An XML DTD defines the legal structure of an XML document.
- **XML Schema**. Like a DTD, an XML schema defines the legal structure of an XML document.
- **XML Namespaces**. Namespaces are a mechanism for differentiating element and attribute names.

  If you require a general introduction to the preceding technologies, consult the XML resources listed in "Related Documents" on page xxxvii of the preface.

**Standards and Specifications**

XML 1.0 is a W3C Recommendation. The C XDK API provides full support for XML 1.0 (Second Edition). You can find the specification for the Second Edition at the following URL:

http://www.w3.org/TR/2000/REC-xml-20001006

The DOM Level 1, Level 2, and Level 3 specifications are W3C Recommendations. The C XDK API provides full support for DOM Level 1 and 2, but no support for Level 3. You can find links to the specifications for all three levels at the following URL:

http://www.w3.org/DOM/DOMTR

SAX is available in version 1.0, which is deprecated, and 2.0. SAX is not a W3C specification. The C XDK API provides full support for both SAX 1.0 and 2.0. You can find the documentation for SAX at the following URL:

http://www.saxproject.org

XML Namespaces are a W3C Recommendation. You can find the specification at the following URL:

http://www.w3.org/TR/REC-xml-names

**See Also:** Chapter 31, "XDK Standards" for a summary of the standards supported by the XDK

**Using the XML Parser for C**

Oracle XML parser for C checks if an XML document is well-formed, and optionally validates it against a DTD. Your application can access the parsed data through the DOM or SAX APIs.

**Overview of the Parser API for C**

The core of the XML parsing API are the XML, DOM, and SAX APIs. Table 18-1 describes the interfaces for these APIs. Refer to *Oracle XML API Reference* for the complete API documentation.
XML Parser for C Datatypes

Refer to Oracle XML API Reference for the complete list of datatypes for the C XDK. Table 18–2 describes the datatypes used in the XML parser for C.

Table 18–2 Datatypes Used in the XML Parser for C

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oratext</td>
<td>String pointer</td>
</tr>
<tr>
<td>xmlctx</td>
<td>Master XML context</td>
</tr>
<tr>
<td>xmlsaxcb</td>
<td>SAX callback structure (SAX only)</td>
</tr>
<tr>
<td>ub4</td>
<td>32-bit (or larger) unsigned integer</td>
</tr>
</tbody>
</table>

---

Using the XML Parser for C
XML Parser for C Defaults

Note the following defaults for the XML parser for C:

- Character set encoding is UTF-8. If all your documents are ASCII, then setting the encoding to US-ASCII increases performance.
- The parser prints messages to stderr unless an error handler is provided.
- The parser checks inputs documents for well-formedness but not validity. You can set the property "validate" to validate the input.

---

**Note:** It is recommended that you set the default encoding explicitly if using only single byte character sets (such as US-ASCII or any of the ISO-8859 character sets) for faster performance than is possible with multibyte character sets such as UTF-8.

---

- The parser conforms to the XML 1.0 specification when processing whitespace, that is, the parser reports all whitespace to the application but indicates which whitespace can be ignored. However, some applications may prefer to set the property "discard-whitespace," which discards all whitespace between an end-element tag and the following start-element tag.

**See Also:**

- Oracle Database XML C API Reference for the DOM, SAX, pull parser, and callback APIs.

XML Parser for C Calling Sequence

Figure 18–1 illustrates the calling sequence for the XML parser for C.
Figure 18–1  XML Parser for C Calling Sequence

Using the XML Parser for C

Using the XML Parser for C 18-5
Using the XML Parser for C: Basic Process

Perform the following steps in your application:

1. Initialize the parsing process with the `XmlCreate()` function. The following sample code fragment is from `DOMNamespace.c`:
   
   ```c
   xmlctx *xctx;
   ...
   xctx = XmlCreate(&ecode, (oratext *) "namespace_xctx", NULL);
   ```

2. Parse the input item, which can be an XML document or string buffer.
   
   If you are parsing with DOM, call the `XmlLoadDom()` function. The following sample code fragment is from `DOMNamespace.c`:
   
   ```c
   xmldocnode *doc;
   ...
   doc = XmlLoadDom(xctx, &ecode, "file", DOCUMENT,
                     "validate", TRUE, "discard_whitespace", TRUE, NULL);
   ```

   If you are parsing with SAX, call the `XmlLoadSax()` function. The following sample code fragment is from `SAXNamespace.c`:
   
   ```c
   xmlerr ecode;
   ...
   ecode = XmlLoadSax(xctx, &sax_callback, &sc, "file", DOCUMENT,
                      "validate", TRUE, "discard_whitespace", TRUE, NULL);
   ```

   If you are using the pull parser, then include the following steps to create the event context and load the document to parse:
   
   ```c
   evctx = XmlEvCreatePPCtx(xctx, &xerr, NULL);
   XmlEvLoadPPDoc(xctx, evctx, "File", input_filenames[i], 0, NULL);
   ```

3. If you are using the DOM interface, then include the following steps:
   
   - Use the `XmlLoadDom()` function to call `XmlDomGetDocElem()`. This step calls other DOM functions, which are typically node or print functions that output the DOM document, as required. The following sample code fragment is from `DOMNamespace.c`:
     
     ```c
     printElements(xctx, XmlDomGetDocElem(xctx, doc));
     ```

   - Invoke the `XmlFreeDocument()` function to clean up any data structures created during the parse process. The following sample code fragment is from `DOMNamespace.c`:
     
     ```c
     XmlFreeDocument(xctx, doc);
     ```

   If you are using the SAX interface, then include the following steps:

   - Process the results of the invocation of `XmlLoadSax()` with a callback function, such as the following
     
     ```c
     xmlsaxcb saxcb = {
       UserStartDocument, /* user's own callback functions */
       UserEndDocument,
       /* ... */
     };
     ```

     ```c
     if (XmlLoadSax(xctx, &saxcb, NULL, "file", "some_file.xml", NULL) != 0)
       /* an error occurred */
     ```
- Register the callback functions. Note that you can set any of the SAX callback functions to `NULL` if not needed.

If you are using the pull parser, iterate over the events using:

```c
cur_event = XmlEvNext(evctx);
```

Use the Get APIs to get information about that event.

4. Use `XmlFreeDocument()` to clean up the memory and structures used during a parse. The program does not free memory allocated for parameters passed to the SAX callbacks or for nodes and data stored with the DOM parse tree until you call `XMLFreeDocument()` or `XMLDestroy()`. The following sample code fragment is from `DOMNamespace.c`:

```c
XmlFreeDocument(xctx, doc);
```

Either return to Step 2 or proceed to the next step.

For the pull parser call `XmlEvCleanPPCtx()` to release memory and structures used during the parse. The application can call `XmlEvLoadPPDoc()` again to parse another document. Or, it can call `XMLEvDestroyPPCtx()` after which the pull parser context cannot be used again.

```c
XmlEvCleanPPCtx(xctx, evctx);
...
XmlEvDestroyPPCtx(xctx, evctx);
```

5. Terminate the parsing process with `XmlDestroy()`. The following sample code fragment is from `DOMNamespace.c`:

```c
(void) XmlDestroy(xctx);
```

If threads fork off somewhere in the sequence of calls between initialization and termination, the application produces unpredictable behavior and results.

You can use the memory callback functions `XML_ALLOC_F` and `XML_FREE_F` for your own memory allocation. If you do, then specify both functions.

See Also:  "What is the XML Pull Parser for C?" on page 18-12

**Running the XML Parser for C Demo Programs**

The `$ORACLE_HOME/xdk/demo/c/` (UNIX) and `%ORACLE_HOME%\xdk\demo\c` (Windows) directories include several XML applications that illustrate how to use the XML parser for C with the DOM and SAX interfaces. Table 18–3 describes the demos.

The `make` utility compiles the source file `fileName.c` to produce the demo program `fileName` and the output file `fileName.out`. The `fileName.std` is the expected output.
You can find documentation that describes how to compile and run the sample programs in the README in the same directory. The basic steps are as follows:

1. Change into the $ORACLE_HOME/xdk/demo/c directory (UNIX) or %ORACLE_HOME%/xdk/demo/c directory (Windows).

2. Make sure that your environment variables are set as described in "Setting C XDK Environment Variables on UNIX" on page 16-3 and "Setting C XDK Environment Variables on Windows" on page 16-6.

3. Run make (UNIX) or make.bat (Windows) at the system prompt. The make utility changes into each demo subdirectory and runs make to do the following:
   a. Compiles the C source files with the cc utility. For example, the Makefile in the $ORACLE_HOME/xdk/demo/c/dom directory includes the following line:

   ```
   $(CC) -o DOMSample $(INCLUDE) $@.c $(LIB)
   ```

   b. Runs each demo program and redirects the output to a file. For example, the Makefile in the $ORACLE_HOME/xdk/demo/c/dom directory includes the following line:

   ```
   ./DOMSample > DOMSample.out
   ```

4. Compare the *.std files to the *.out files for each program. The *.std file contains the expected output for each program. For example, DOMSample.std contains the expected output from running DOMSample.
Using the C XML Parser Command-Line Utility

The xml utility, which is located in $ORACLE_HOME/bin (UNIX) or %ORACLE_HOME%\bin (Windows), is a command-line interface that parses XML documents. It checks for both well-formedness and validity.

To use xml ensure that your environment is set up as described in “Setting C XDK Environment Variables on UNIX” on page 16-3 and “Setting C XDK Environment Variables on Windows” on page 16-6.

Use the following syntax on the command line to invoke xml. Use xml.exe for Windows:

\texttt{xml [options] [document URI]}
\texttt{xml -f [options] [document filespec]}

Table 18–4 describes the command-line options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>Checks well-formedness, but performs no validation.</td>
</tr>
<tr>
<td>-e encoding</td>
<td>Specifies default input file encoding (&quot;incoding&quot;).</td>
</tr>
<tr>
<td>-E encoding</td>
<td>Specifies DOM/SAX encoding (&quot;outcoding&quot;).</td>
</tr>
<tr>
<td>-f file</td>
<td>Interprets the file as filespec, not URI.</td>
</tr>
<tr>
<td>-G xptr_exps</td>
<td>Evaluates XPointer scheme examples given in a file.</td>
</tr>
<tr>
<td>-h</td>
<td>Shows usage help and basic list of command-line options.</td>
</tr>
<tr>
<td>-hh</td>
<td>Shows complete list command-line options.</td>
</tr>
<tr>
<td>-i n</td>
<td>Specifies the number of times to iterate the XSLT processing.</td>
</tr>
<tr>
<td>-l language</td>
<td>Specifies the language for error reporting.</td>
</tr>
<tr>
<td>-n</td>
<td>Traverses the DOM and reports the number of elements, as shown in the following sample output:</td>
</tr>
<tr>
<td></td>
<td>\textbf{TOTAL} \textbf{3 \times 60 = 180}</td>
</tr>
<tr>
<td>-o XSLoutfile</td>
<td>Specifies the output file of the XSLT processor.</td>
</tr>
<tr>
<td>-p</td>
<td>Prints the document/DTD structures after the parse. For example, the root element \texttt{&lt;greeting&gt;hello&lt;/greeting&gt;} is printed as:</td>
</tr>
<tr>
<td></td>
<td>\texttt{+---ELEMENT greeting}</td>
</tr>
<tr>
<td></td>
<td>\texttt{  +---PCDATA &quot;hello&quot;}</td>
</tr>
<tr>
<td>-P</td>
<td>Prints the document from the root element. For example, the root element \texttt{&lt;greeting&gt;hello&lt;/greeting&gt;} is printed as:</td>
</tr>
<tr>
<td></td>
<td>\texttt{&lt;greeting&gt;hello&lt;/greeting&gt;}</td>
</tr>
<tr>
<td>-PP</td>
<td>Prints from the root node (DOC) and includes the XML declaration.</td>
</tr>
<tr>
<td>-PE encoding</td>
<td>Specifies the encoding for -P or -PP output.</td>
</tr>
<tr>
<td>-PX</td>
<td>Includes the XML declaration in the output.</td>
</tr>
<tr>
<td>-s stylesheet</td>
<td>Specifies the XSLT stylesheet.</td>
</tr>
</tbody>
</table>
Using the XML Parser Command-Line Utility: Example

You can test xml on the various XML files located in $ORACLE_HOME/xdk/demo/c. Example 18–1 displays the contents of NSExample.xml.

**Example 18–1 NSExample.xml**

```xml
<!DOCTYPE doc [
<!ELEMENT doc (child*)>
<!ATTLIST doc xmlns:nsprefix CDATA #IMPLIED>
<!ATTLIST doc xmlns CDATA #IMPLIED>
<!ATTLIST doc nsprefix:a1 CDATA #IMPLIED>
<!ELEMENT child (#PCDATA)>]

<doc nsprefix:a1 = "v1" xmlns="http://www.w3c.org" xmlns:nsprefix="http://www.oracle.com">
  <child>
    This element inherits the default Namespace of doc.
  </child>
</doc>
```

You can parse this file, count the number of elements, and display the DOM tree as shown in the following example:

```bash
xml -np NSExample.xml > xml.out
```

The output is shown in the following example:

**Example 18–2 xml.out**

```plaintext
ELEMENT       2
PCDATA       1
DOC       1
DTD       1
ELEMENTDECL       2
ATTRDECL       3
TOTAL      10 * 112 = 1120

+-++ELEMENT doc [nsprefix:a1='v1*', xmlns='http://www.w3c.org*', xmlns:nsprefix='http://www.oracle.com']
  +--ELEMENT child
    +---PCDATA 'This element inherits the default Namespace of doc.'
```

### Table 18–4 (Cont.) C XML Parser Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-v</code></td>
<td>Displays the XDK parser version and then exits.</td>
</tr>
<tr>
<td><code>-V var value</code></td>
<td>Tests top-level variables in CXSLT.</td>
</tr>
<tr>
<td><code>-w</code></td>
<td>Preserves all whitespace.</td>
</tr>
<tr>
<td><code>-W</code></td>
<td>Stops parsing after a warning.</td>
</tr>
<tr>
<td><code>-x</code></td>
<td>Exercises the SAX interface and prints the document, as shown in the following sample output:</td>
</tr>
<tr>
<td></td>
<td>StartDocument</td>
</tr>
<tr>
<td></td>
<td>XMLDECL version='1.0' encoding=FALSE</td>
</tr>
<tr>
<td></td>
<td>&lt;greeting&gt;</td>
</tr>
<tr>
<td></td>
<td>'hello'</td>
</tr>
<tr>
<td></td>
<td>&lt;/greeting&gt;</td>
</tr>
<tr>
<td></td>
<td>EndDocument</td>
</tr>
</tbody>
</table>

Using the XML Parser for C
Using the DOM API for C

This section contains the following topics:

- Controlling the Data Encoding of XML Documents for the C API
- Using NULL-Terminated and Length-Encoded C API Functions
- Handling Errors with the C API

Controlling the Data Encoding of XML Documents for the C API

XML data occurs in many encodings. You can control the XML encoding in the following ways:

- Specify a default encoding to assume for files that are not self-describing
- Specify the presentation encoding for DOM or SAX
- Re-encode when a DOM is serialized

Input XML data is always encoded. Some encodings are entirely self-describing, such as UTF-16, which requires a specific BOM before the start of the actual data. The XMLDecl or MIME header of the document can also specify an encoding. If the application cannot determine the specific encoding, then it applies the default input encoding. If you do not provide a default, then the application assumes UTF-8 on ASCII platforms and UTF-E on EBCDIC platforms.

The API makes a provision for cases when the encoding data of the input document is corrupt. For example, suppose an ASCII document with an XMLDecl of encoding=ascii is blindly converted to EBCDIC. The new EBCDIC document contains (in EBCDIC) an XMLDecl that incorrectly claims the document is ASCII. The correct behavior for a program that is re-encoding XML data is to regenerate but not convert the XMLDecl. The XMLDecl is metadata, not data itself. This rule is often ignored, however, which results in corrupt documents. To work around this problem, the API provides an additional flag that enables you to forcibly set the input encoding, thereby overcoming an incorrect XMLDecl.

The precedence rules for determining input encoding are as follows:

1. Forced encoding as specified by the user

Caution: Forced encoding can result in a fatal error if there is a conflict. For example, the input document is UTF-16 and starts with a UTF-16 BOM, but the user specifies a forced UTF-8 encoding. In this case, the parser objects about the conflict.

2. Protocol specification (HTTP header, and so on)
3. XMLDecl specification
4. User's default input encoding
5. The default, which is UTF-8 on ASCII platforms or UTF-E on EBCDIC platforms

After the application has determined the input encoding, it can parse the document and present the data. You are allowed to choose the presentation encoding; the data is in that encoding regardless of the original input encoding.
When an application writes back a DOM in serialized form, it can choose at that time to re-encode the presentation data. Thus, the you can place the serialized document in any encoding.

Using NULL-Terminated and Length-Encoded C API Functions

The native string representation in C is NULL-terminated. Thus, the primary DOM interface takes and returns NULL-terminated strings. When stored in table form, however, Oracle XML DB data is not NULL-terminated but length-encoded. Consequently, the XDK provides an additional set of length-encoded APIs for the high-frequency cases to improve performance. In particular, the DOM functions in Table 18–5 have dual APIs.

<table>
<thead>
<tr>
<th>NULL-Terminated API</th>
<th>Length-Encoded API</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmlDomGetNodeName()</td>
<td>XmlDomGetNodeNameLen()</td>
</tr>
<tr>
<td>XmlDomGetNodeLocal()</td>
<td>XmlDomGetNodeLocalLen()</td>
</tr>
<tr>
<td>XmlDomGetNodeURI()</td>
<td>XmlDomGetNodeURILen()</td>
</tr>
<tr>
<td>XmlDomGetNodeValue()</td>
<td>XmlDomGetNodeValueLen()</td>
</tr>
<tr>
<td>XmlDomGetAttrName()</td>
<td>XmlDomGetAttrNameLen()</td>
</tr>
<tr>
<td>XmlDomGetAttrLocal()</td>
<td>XmlDomGetAttrLocalLen()</td>
</tr>
<tr>
<td>XmlDomGetAttrURI()</td>
<td>XmlDomGetAttrURILen()</td>
</tr>
<tr>
<td>XmlDomGetAttrValue()</td>
<td>XmlDomGetAttrValueLen()</td>
</tr>
</tbody>
</table>

Handling Errors with the C API

The C API functions typically either return a numeric error code (0 for success, nonzero on failure), or pass back an error code through a variable. In all cases, the API stores error codes. Your application can retrieve the most recent error by calling the XmlDomGetLastError() function.

By default, the functions output error messages to stderr. However, you can register an error message callback at initialization time. When an error occurs, the application invokes the registered callback and does not print an error.

Using the SAX API for C

To use SAX, initialize an xmlsaxcb structure with function pointers and pass it to XmlLoadSax(). You can also include a pointer to a user-defined context structure, which you pass to each SAX function.

See Also: Oracle Database XML C API Reference for the SAX callback structure

What is the XML Pull Parser for C?

The XML Pull Parser is an implementation of the XML Events interface.

The XML Pull Parser and the SAX parser are similar, but using the Pull Parser, the application (consumer) drives the events, while in SAX, the parser (producer) drives the events. Both the XML Pull Parser and SAX represent the document as a sequence of events, with start tags, end tags, and comments.
What is the XML Pull Parser for C?

XML Pull Parser gives control to the application by exposing a simple set of APIs and an underlying set of events. Methods such as XmlEvNext allow an application to ask for (or pull) the next event, rather than handling the event in a callback, as in SAX. Thus, the application has more procedural control over XML processing. Also, the application can decide to stop further processing, unlike a SAX application, which parses the entire document.

This section contains the following topics:

- Using Basic XML Pull Parsing Capabilities
- Parsing Multiple XML Documents
- ID Callback
- Error Handling for the XML Pull Parser
- Sample Pull Parser Application

Using Basic XML Pull Parsing Capabilities

To use the XML Pull Parser, your application must do the following, in the order given:

1. Call XmlCreate to initialize the XML meta-context.
2. Initialize the Pull Parser context with a call to the XmlEvCreatePPCtx function, which creates and returns the event context.
   - The XmlEvCreatePPCtx function supports all the properties supported by XmlLoadDom and XmlLoadSax, plus some additional ones.
   - The XmlEvCreatePPCtx and XmlEvCreatePPCtxVA functions are fully implemented.
3. Ensure that the event context is passed to all subsequent calls to the Pull Parser.
4. Terminate the Pull Parser context by calling the XmlEvDestoryPPCtx function, to clean up memory.
5. Destroy the XML meta-context by calling the XmlDestoryCtx function.

XML Event Context

Example 18–3 shows the structure of the event context.

Example 18–3  XML Event Context

typedef struct {
    void *ctx_xmllevctx;          /* implementation specific context */
    xmllevdisp *disp_xmllevctx;   /* dispatch table */
    ub4 checkword_xmllevctx;      /* checkword for integrity check */
    ub4 flags_xmllevctx;          /* mode; default: expand_entity */
    struct xmllevctx *input_xmllevctx;  /* input xmllevctx; chains the XML Event context */
} xmllevctx;

About the XML Event Context

Each XML Pull Parser is allowed to create its own context and implement its own API functions.

- Dispatch Table
The dispatch table, disp_xmlevctx, contains one pointer for each API function, except for the XmlEvCreatePPCtx, XmlEvCreatePPCtxVA, XmlEvDestoryPPCtx, XmlEvLoadPPDoc, and XmlEvCleanPPCtx functions.

When the event context is created, the pointer disp_xmlevctx is initialized with the address of that static table.

- Implementation-Specific Event Context
  The field ctx_xmlevctx must be initialized with the address of the context specific to this invocation of the particular implementation. The implementation-specific event context is of type *void, so that it can differ for different applications.

- Input Event Context
  Each Pull Parser can specify an input event context, xmlevctx. This field enables the parser to chain multiple event producers. As a result, if a dispatch function is specified as NULL in a context, the application uses the next non-null dispatch function in the chain of input event contexts. The base xmlevctx must ensure that all dispatch function pointers are non-null.

Parsing Multiple XML Documents

After creating and initializing the XML Event Context, the application can parse multiple documents with repeated calls to XmlEvLoadPPDoc and XmlEvCleanPPCtx. These functions are fully implemented.

Note that the properties defined by the application during the XML Event Context creation cannot be changed for each call to the XmlLoadPPDoc function. If you want to change the properties, destroy the event context and re-create it.

After XmlEvCleanPPCtx cleans up the internal structure of the current parser, the event context can be re-used to parse another document.

ID Callback

You can provide a callback to convert text-based names to 8-byte IDs.

**Callback Function Signature**

typedef sb8 (*xmlev_id_cb_funcp)( void *ctx , ub1 type, ub1 *token, ub4 tok_len, sb8 nmspid, boolean isAttribute);

**Return Value**

sb8: an 8-byte ID.

**Arguments**

- *ctx: The implementation context.
- type: The type, which is indicated by the following enumeration:

  typedef enum
  {
    XML_EVENT_ID_URI,
    XML_EVENT_ID_QNAME,
  }xmlevidtype;

- *token and tok_len: The actual text to be converted.
- nmspid: The namespace ID.
- isAttribute: A Boolean value indicating an attribute.

Internally, the XmlEvGetTagId and XmlEvGetAttrID APIs call this callback twice, once to fetch the namespace ID and once to fetch the actual ID of the tag or the attribute QName.

The XmlEvGetTagUriID and XmlEvGetAttrUriID functions invoke this callback once to get the ID of the corresponding URI.

If a callback is not supplied, an error XML_ERR_EVENT_NOIDCBK is returned when these APIs are used.

**Error Handling for the XML Pull Parser**

The following sections describe error handling for the XML Pull Parser.

**Parser Errors**

The XML Pull Parser returns the message XML_EVENT_FATAL_ERROR when it throws an error because the input document is malformed. The XmlEvGetError function is provided to get the error number and message.

Note that during the XmlEvCreatePPCtx operation, any error handler supplied by the application during XmlCreate is overridden. The application must call the XmlErrSetHandler function after the XmlEvDestroyPPCtx operation to restore the original callback.

**Programming Errors**

To handle programmatic errors, XDK provides a callback that the application can supply when creating an event context. This callback is invoked when the application makes a call to an illegal API. The callback signature is as follows:

```c
typedef void (*xmlev_err_cb_funcp)(xmlctx *xctx, xmlevctx *evctx, xmlevtype cur_event);
```

An example of an illegal API call is:

XmlEvGetName cannot be called for the XML_EVENT_CHARACTERS event.

**Sample Pull Parser Application**

This section contains a sample pull parser application, a document to be parsed, and a list of the events that the application generates from the document.

**An XML Pull Parser Sample Application**

```c
#include "xml.h"
#include "xmlev.h"
...
xmctx *xctx;
xmlevctx *evctx;
if (!xctx = XmlCreate(&xerr, (oratext *) "test"))
{
    printf("Failed to create XML context, error %u\n", (unsigned) xerr);
    return -1;
}
...
if(!(evctx = XmlEvCreatePPCtx(xctx, &xerr, NULL)))
{
    printf("Failed to create EVENT context, error %u\n", (unsigned) xerr);
    return -1;
}
for(i = 0; i < numDocs; i++)
{
    if (xerr = XmlEvLoadPPDoc(xctx, evctx, "file", input_filenames[i], 0, NULL)
    {
        printf("Failed to load the document, error %u\n", (unsigned) xerr);
        return -1;
    }
    ...
    for(;;)
    {
        xmlевtype cur_event;
        cur_event = XmlEvNext(evctx);
        switch(cur_event)
        {
            case XML_EVENT_FATAL_ERROR:
                XmlEvGetError(evctx, (oratext **)&errmsg);
                printf("Error %s\n", errmsg);
                return;
            case XML_EVENT_START_ELEMENT:
                printf("<?s >", XmlEvGetName0(evctx));
                break;
            case XML_EVENT_END_DOCUMENT:
                printf("<?s >", XmlEvGetName0(evctx));
                return;
        }
    }
    XmlEvCleanPPCtx(xctx, evctx);
}
XmlEvDestroyPPCtx(xctx, evctx);
XmlDestroy(xctx);

Sample Document

Example 18–5 Sample Document to Parse
<!DOCTYPE doc [
<!ENTITY ent SYSTEM "file:attendees.txt">
<!ELEMENT doc ANY>
<!ELEMENT meeting (topic, date, publishAttendees)>
<!ELEMENT publishAttendees (#PCDATA)>
<!ELEMENT topic (#PCDATA)>
<!ELEMENT date (#PCDATA)>
]>
<!-- Begin Document -->
<doc>
<!-- Info about the meeting -->
<meeting>
 <topic>Group meeting</topic>
 <date>April 25, 2005</date>
 <publishAttendees>&ent;</publishAttendees>
</meeting>
</doc>
<!-- End Document -->
Events Generated by XML Pull Parser Sample Application

This is the sequence of events generated when the attribute events property is FALSE and expand entities properties is TRUE.

Example 18–6 Events Generated by Parsing a Sample Document

XML_EVENT_START_DOCUMENT
XML_EVENT_START_DTD
XML_EVENT_PE_DECLARATION
XML_EVENT_ELEMENT_DECLARATION
XML_EVENT_ELEMENT_DECLARATION
XML_EVENT_ELEMENT_DECLARATION
XML_EVENT_ELEMENT_DECLARATION
XML_EVENT_ELEMENT_DECLARATION
XML_EVENT_ELEMENT_DECLARATION
XMLEVENT_START_DTD
XML_EVENT_COMMENT
XML_EVENT_START_ELEMENT
XML_EVENT_SPACE
XML_EVENT_COMMENT
XML_EVENT_SPACE
XML_EVENT_START_ELEMENT
XML_EVENT_START_ELEMENT
XML_EVENT_CHARACTERS
XML_EVENT_END_ELEMENT
XML_EVENT_START_ELEMENT
XML_EVENT_CHARACTERS
XML_EVENT_END_ELEMENT
XML_EVENT_START_ELEMENT
XML_EVENT_START_ENTITY
XML_EVENT_CHARACTERS
XML_EVENT_END_ENTITY
XML_EVENT_END_ELEMENT
XML_EVENT_END_ELEMENT
XML_EVENT_SPACE
XML_EVENT_END_ELEMENT
XML_EVENT_COMMENT
XML_EVENT_END_DOCUMENT

Using OCI and the XDK C API

This section describes calling the XDK C functions from Oracle Call Interface (OCI).

Using XMLType Functions and Descriptions

You can use the C API for XML for XMLType columns in the database. An OCI program can access XML data stored in a table by initializing the values of OCI handles such as the following:

- Environment handle
- Service handle
- Error handle
- Optional parameters

The program can pass these input values to the function OCIXmlDbInitXmlCtx(), which returns an XML context. After the program makes calls to the C API, the function OCIXmlDbFreeXmlCtx() frees the context.

Table 18–6 describes a few of the functions for XML operations.
Initializing an XML Context for XML DB

An XML context is a required parameter in all the C DOM API functions. This opaque context encapsulates information pertaining to data encoding, error message language, and so on. The contents of this XML context are different for XDK applications and for Oracle XML DB applications.

Caution: Do not use an XML context for XDK in an XML DB application, or an XML context for XML DB in an XDK application.

For Oracle XML DB, the two OCI functions that initialize and free an XML context have the following prototypes:

```c
xmlctx *OCIXmlDbInitXmlCtx (OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp, ocixmldbparam *params, ub4 num_params);
void OCIXmlDbFreeXmlCtx (xmlctx *xctx);
```

See Also:
- Oracle Call Interface Programmer’s Guide for reference material on the functions
- Oracle Call Interface Programmer’s Guide for a discussion about OCI support for XML
- Oracle Database XML C API Reference for reference information on the DOM APIs

Creating XMLType Instances on the Client

You can construct new XMLType instances on the client by using the XmlLoadDom() calls. Follow these basic steps:

1. You first have to initialize the xmlctx, as illustrated in the example in "Using the DOM API for C" on page 18-11.
2. You can construct the XML data itself from the following sources:

---

Table 18–6  XMLType Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmlCreateDocument()</td>
<td>Create empty XMLType instance</td>
</tr>
<tr>
<td>XmlLoadDom() and so on</td>
<td>Create from a source buffer</td>
</tr>
<tr>
<td>XmlXPathEvalexpr() and family</td>
<td>Extract an XPath expression</td>
</tr>
<tr>
<td>XmlXslProcess() and family</td>
<td>Transform using an XSLT stylesheet</td>
</tr>
<tr>
<td>XmlXPathEvalexpr() and family</td>
<td>Check if an XPath exists</td>
</tr>
<tr>
<td>XmlDomIsSchemaBased()</td>
<td>Is document schema-based?</td>
</tr>
<tr>
<td>XmlDomGetSchema()</td>
<td>Get schema information</td>
</tr>
<tr>
<td>XmlDomGetNodeURI()</td>
<td>Get document namespace</td>
</tr>
<tr>
<td>XmlSchemaValidate()</td>
<td>Validate using schema</td>
</tr>
<tr>
<td>Cast (void *) to (xmldocnode *)</td>
<td>Obtain DOM from XMLType</td>
</tr>
<tr>
<td>Cast (xmldocnode *) to (void *)</td>
<td>Obtain XMLType from DOM</td>
</tr>
</tbody>
</table>

Caution: Do not use an XML context for XDK in an XML DB application, or an XML context for XML DB in an XDK application.
Using OCI and the XDK C API

- User buffer
- Local file
- URI

The return value from these is an (xmldocnode *), which you can use in the rest of the common C API.

3. Finally, you can cast the (xmldocnode *) to a (void *) and directly provide it as the bind value if required.

You can construct empty XMLType instances by using the XmlCreateDocument() call. This function would be equivalent to an OCIObjectNew() for other types. You can operate on the (xmldocnode *) returned by the preceding call and finally cast it to a (void *) if it needs to be provided as a bind value.

Operating on XML Data in the Database Server

You can operate on XML data in Oracle Database by means of OCI statement calls. You can bind and define XMLType values using xmldocnode and use OCI statements to select XML data from the database. You can use this data directly in the C DOM functions. Similarly, you can bind the values directly to SQL statements.

Using OCI and the XDK C API: Examples

Example 18–7 illustrates how to construct a schema-based document with the DOM API and save it to the database. Note that you must include the header files xml.h and ocixmlDb.h.

**Example 18–7 Constructing a Schema-Based Document with the DOM API**

```c
#include <xml.h>
#include <ocixmlDb.h>
static oratext tlpxml_test_sch[] = "<TOP xmlns='example1.xsd'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xsi:schemaLocation='example1.xsd example1.xsd'/>";

void example1()
{
    OCIEnv *envhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCIStmt *stmthp;
    OCIDuration dur;
    OCIType *xmltdo;
    xmldocnode  *doc;
    ocixmlDbParam params[1];
    xmlnode *quux, *foo, *foo_data;
    xmlerr       err;

    /* Initialize envhp, svchp, errhp, dur, stmthp */
    /* ........ */

    /* Get an xml context */
    params[0].name_ocixmlDbParam = XCTXINIT_OCIDUR;
    params[0].value_ocixmlDbParam = &dur;
    xctx = OCIXmlDbInitXmlCtx(envhp, svchp, errhp, params, 1);

    /* Start processing */
```

Using the XML Parser for C 18-19
Using OCI and the XDK C API

printf("Supports XML 1.0: %s\n",
XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ? "YES" : "NO");
/* Parsing a schema-based document */
if (!(doc = XmlLoadDom(xctx, &err, "buffer", tlpxml_test_sch,
"buffer_length", sizeof(tlpxml_test_sch)-1,
"validate", TRUE, NULL)))
{
printf("Parse failed, code %d\n");
return;
}
/* Create some elements and add them to the document */
top = XmlDomGetDocElem(xctx, doc);
quux = (xmlnode *) XmlDomCreateElem(xctx ,doc, (oratext *) "QUUX");
foo = (xmlnode *) XmlDomCreateElem(xctx, doc, (oratext *) "FOO");
foo_data = (xmlnode *) XmlDomCreateText(xctx, doc, (oratext *)"foo's data");
foo_data = XmlDomAppendChild(xctx, (xmlnode *) foo, (xmlnode *) foo_data);
foo = XmlDomAppendChild(xctx, quux, foo);
quux = XmlDomAppendChild(xctx, top, quux);
XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);
XmlSaveDom(xctx, &err, doc, "stdio", stdout, NULL);
/* Insert the document to my_table */
ins_stmt = "insert into my_table values (:1)";
status = OCITypeByName(envhp, errhp, svchp, (const text *) "SYS",
(ub4) strlen((char *)"SYS"), (const text *) "XMLTYPE",
(ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
(ub4) 0, dur, OCI_TYPEGET_HEADER,
(OCIType **) &xmltdo)) ;
if (status == OCI_SUCCESS)
{
exec_bind_xml(svchp, errhp, stmthp, (void *)doc, xmltdo, ins_stmt));
}
/* free xml ctx */
OCIXmlDbFreeXmlCtx(xctx);
}
/*--------------------------------------------------------*/
/* execute a sql statement which binds xml data */
/*--------------------------------------------------------*/
sword exec_bind_xml(svchp, errhp, stmthp, xml, xmltdo, sqlstmt)
OCISvcCtx *svchp;
OCIError *errhp;
OCIStmt *stmthp;
void *xml;
OCIType *xmltdo;
OraText *sqlstmt;
{
OCIBind *bndhp1 = (OCIBind *) 0;
OCIBind *bndhp2 = (OCIBind *) 0;
sword status = 0;
OCIInd ind = OCI_IND_NOTNULL;
OCIInd *indp = &ind;
if(status = OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,

18-20 Oracle XML Developer's Kit Programmer's Guide


Example 18–8 illustrates how to get a document from the database and modify it with the DOM API.

**Example 18–8  Modifying a Database Document with the DOM API**

```c
#include <xml.h>
#include <ocixmldb.h>
sword example2() {

  OCIEnv *envhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  OCIStmt *stmthp;
  OCIDuration dur;
  OCIType *xmltdo;
  xmldocnode *doc;
  xmlnodelist *item_list; ub4 ilist_l;
  ocixmldbparam params[1];
  text *sel_xml Stmt = (text *)"SELECT xml_col FROM my_table";
  ub4 xmlsize = 0;
  sword status = 0;
  OCIDefine *defnp = (OCIDefine *)0;

  /* Initialize envhp, svchp, errhp, dur, stmthp */
  /* ... */

  /* Get an xml context */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
  params[0].value_ocixmldbparam = &dur;
  xctx = OCIXmlDbInitXmlCtx(envhp, svchp, errhp, params, 1);

  /* Start processing */
  if(status = OCITypeByName(envhp, errhp, svchp, (const text *)"SYS",
      (ub4) strlen((char *)"SYS"), (const text *)"XMLTYPE",
      (ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
      (ub4) OCI_DEFAULT)) {
    return OCI_ERROR;
  }

  if(status = OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 1, (dvoid *)0,
      (ub4) 0, SQLT_NTY, (dvoid *)0, (ub2 *)0,
      (ub4) 0, (ub4) 0, (ub4) OCI_DEFAULT)) {
    return OCI_ERROR;
  }

  if(status = OCIBindObject(bndhp1, errhp, (CONST OCIType *) xmltdo,
      (dvoid **) &xml, (ub4 *)0, (dvoid **) &indp, (ub4 *)0)) {
    return OCI_ERROR;
  }

  if(status = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
      (CONST OCISnapshot*)0, (OCISnapshot*)0, (ub4) OCI_DEFAULT)) {
    return OCI_ERROR;
  }

  return OCI_SUCCESS;
}
```
Using OCI and the XDK C API

```c
{ub4} 0, dur, OCI_TYPEGET_HEADER,
   (OCIType **) xmltdo_p}) {
   return OCI_ERROR;
}

if(!(*xmltdo_p)) {
   printf("NULL tdo returned\n");
   return OCI_ERROR;
}

if(status = OCIStmtPrepare(stmthp, errhp, (OraText *) selstmt,
   {ub4}strlen((char *) selstmt),
   {ub4} OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT)) {
   return OCI_ERROR;
}

if(status = OCIDefineByPos(stmthp, &defnp, errhp, {ub4} 1, (dvoid *) 0,
   {sb4} 0, SQLT_NTY, (dvoid *) 0, {ub2} 0,
   {ub2} 0, {ub4} OCI_DEFAULT)) {
   return OCI_ERROR;
}

if(status = OCIDefineObject(defnp, errhp, (OCIType *) *xmltdo_p,
   dvoid **) &doc,
   &xmlsize, (dvoid **) 0, (ub4 *) 0)) {
   return OCI_ERROR;
}

if(status = OCIStmtExecute(svchp, stmthp, errhp, {ub4} 1, {ub4} 0,
   (CONST OCISnapshot*) 0, (OCISnapshot*) 0, {ub4} OCI_DEFAULT)) {
   return OCI_ERROR;
}

/* We have the doc. Now we can operate on it */
printf("Getting Item list...\n");

item_list = XmlDomGetElemsByTag(xctx, (xmlelemnode *) elem, (orastr *)"Item");
ilist_l   = XmlDomGetNodeListLength(xctx, item_list);
printf(" Item list length = %d 
", ilist_l);

for (i = 0; i < ilist_l; i++)
{
   elem = XmlDomGetNodeListItem(xctx, item_list, i);
   printf("Elem Name:%s\n", XmlDomGetNodeName(xctx, fragelem));
   XmlDomRemoveChild(xctx, fragelem);
}

XmlSaveDom(xctx, &err, doc, "stdio", stdout, NULL);

/* free xml ctx */
OCIXmlDbFreeXmlCtx(xctx);

return OCI_SUCCESS;
```
Using Binary XML for C

This chapter contains this topic.

- Introduction to Binary XML for C

Introduction to Binary XML for C

Client-side processing of XML data can use either XMLType data stored in the database, including data in binary XML format, or transient data that is not in the database.

Prerequisites

This chapter assumes that you are familiar with the XML Parser for C, the basic concepts of binary XML, and the OCI (Oracle Call Interface). For this release, only the OCI API can be used for programming in C with binary XML. A discussion of how to use OCI support for XML is found in the Oracle Call Interface Programmer’s Guide, section "OCI Support for XML."

See Also:

- Chapter 18, "Using the XML Parser for C"
- Chapter 5, "Using Binary XML for Java"
- Oracle XML DB Developer’s Guide
- Oracle Call Interface Programmer’s Guide

Binary XML Storage Format

Binary XML was introduced in 11g Release 1 (11.1). Binary XML is an optimized format for XML. It includes encoding and decoding of XML documents, from text to binary and binary to text.

Binary XML is XML Schema-aware encoding of XML data, but binary XML can also be used for XML data that is not based on an XML schema.

A Binary XML processor is a component that processes and transforms binary XML format into text and XML text into binary XML format.

The mid-tier and client tiers can produce, consume, and process XML in binary XML format. The C application fetches data from the XML DB repository, performs updates on the XML using DOM, and stores it back in the database. Or an XML document is created or input on the client and XSLT, XQuery, and other utilities can be used on it. Then the output XML is saved in XML DB. Further details of concepts and reference pages for OCI functions are described in the Oracle Call Interface Programmer’s Guide.
Using the XML Schema Processor for C

This chapter contains these topics:

- Oracle XML Schema Processor for C
- Using the C XML Schema Processor Command-Line Utility
- XML Schema Processor for C Usage Diagram
- How to Run XML Schema for C Sample Programs
- What is the Streaming Validator?

**Note:** Use the new unified C API for new XDK and Oracle XML DB applications. The old C functions are deprecated and supported only for backward compatibility, but will not be enhanced. They will be removed in a future release.

The new C API is described in "Overview of the Unified C API" on page 16-10.

### Oracle XML Schema Processor for C

The XML Schema processor for C is a companion component to the XML parser for C that allows support for simple and complex datatypes in XML applications.

The XML Schema processor for C supports the W3C XML Schema Recommendation. This makes writing custom applications that process XML documents straightforward, and means that a standards-compliant XML Schema processor is part of the XDK on every operating system where Oracle is ported.

The XML Schema processor enables validation of XML and retrieval of metadata. It can be called by itself or through the XML Parser for C.

**See Also:** Chapter 4, "XML Parsing for Java", for more information about XML Schema and why you would want to use XML Schema.

### Oracle XML Schema for C Features

XML Schema processor for C has the following features:

- Supports simple and complex types
- Built on XML parser for C
- Supports the W3C XML Schema Recommendation
Standards Conformance

The Schema Processor conforms to the following standards:

- W3C recommendation for Extensible Markup Language (XML) 1.0
- W3C recommendation for Document Object Model Level 1.0
- W3C recommendation for Namespaces in XML
- W3C recommendation for XML Schema

XML Schema Processor for C: Supplied Software

Table 20–1 lists the supplied files and directories for this release.

<table>
<thead>
<tr>
<th>Directory and Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>schema processor executable, schema</td>
</tr>
<tr>
<td>lib</td>
<td>XML/XSL/Schema &amp; support libraries</td>
</tr>
<tr>
<td>nls/data</td>
<td>Globalization Support data files</td>
</tr>
<tr>
<td>xdk/demo/c/schema</td>
<td>example usage of the Schema processor</td>
</tr>
<tr>
<td>xdk/include</td>
<td>header files</td>
</tr>
<tr>
<td>xdk/mesg</td>
<td>error message files</td>
</tr>
<tr>
<td>xdk/readme.html</td>
<td>introductory file</td>
</tr>
</tbody>
</table>

Table 20–2 lists the included libraries in directory lib.

<table>
<thead>
<tr>
<th>Included Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libxml10.a</td>
<td>XML parser, XSLT processor, XML Schema processor</td>
</tr>
<tr>
<td>libcxml10.a</td>
<td>CORE functions</td>
</tr>
<tr>
<td>libnls10.a</td>
<td>Globalization Support</td>
</tr>
</tbody>
</table>

Using the C XML Schema Processor Command-Line Utility

XML Schema processor for C can be called as an executable by invoking bin/schema in the install area. This takes two arguments:

- XML instance document
- Optionally, a default schema

The XML Schema processor for C can also be invoked by writing code using the supplied APIs. The code must be compiled using the headers in the include subdirectory and linked against the libraries in the lib subdirectory. See Makefile in the xdk/demo/c/schema subdirectory for details on how to build your program.

Error message files in different languages are provided in the mesg subdirectory.
**XML Schema Processor for C Usage Diagram**

Figure 20–1 describes the calling sequence for the XML Schema processor for C, as follows:

1. The initialize call is invoked once at the beginning of a session; it returns a schema context which is used throughout the session.
2. Schema documents to be used in the session are loaded in advance.
3. The instance document to be validated is first parsed with the XML parser.
4. The top of the XML element subtree for the instance is then passed to the schema validate function.
5. If no explicit schema is defined in the instance document, any pre-loaded schemas will be used.
6. More documents can then be validated using the same schema context.
7. When the session is over, the Schema tear-down function is called, which releases all memory allocated for the loaded schemas.

**How to Run XML Schema for C Sample Programs**

The directory `xdk/demo/c/schema` contains sample XML Schema applications that illustrate how to use Oracle XML Schema processor with its API. Table 20–3 lists the provided sample files.
What is the Streaming Validator?

The streaming validator was introduced in 11g Release 1 (11.1). It uses XML Events, which is a representation of an XML document that is similar to SAX Events. XML events has a start tag, end tag, and comment. The producer drives the SAX events and the consumer drives the XML events. The streaming validator shares software with the older schema validator and derives most functionality from it. Memory overhead is less than for the DOM representation used in the older validator. Only one pass is made over the document.

There are two modes of streaming validation:

- Transparent mode - events are returned to the application.
- Opaque mode - events are not returned to the application but an error indicating success or failure of the document validation process is returned.

Before document validation, the regular validation context must be created, and the relevant schema must be loaded using this context. Then XML event context for pull parser (or for another event producer) must be created. This event context is then given to the streaming validator, so that it is able to request events from the producer.

Passing in a schema DOM to the `XmlSchemaLoad` API is also supported.

### Using Transparent Mode

An application starts with a call to `XmlEvCreateSVCtx()`. This call creates and returns an event context of type `xmlctx`, which must be passed on all subsequent calls to the streaming validator. The event context created must be terminated by a call to `XmlEvDestroyCtx()`.

After creation of the event context, the application repeatedly advances validation to the next event by issuing calls to `XmlEvNext()`, which returns the type of the next event. Additional API interfaces allow the application to retrieve information relevant to the last event.

---

### Table 20–3  XML Schema for C Samples Provided

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile</td>
<td>Makefile to build the sample programs and run them, verifying correct output.</td>
</tr>
<tr>
<td>xsdtest.c</td>
<td>Program which invokes the XML Schema for C API</td>
</tr>
<tr>
<td>car.{xsd,xml,std}</td>
<td>Sample schema, instance document, and expected output respectively, after running xsdtest on them.</td>
</tr>
<tr>
<td>aq.{xsd,xml,std}</td>
<td>Second sample schema, instance document, and expected output respectively, after running xsdtest on them.</td>
</tr>
<tr>
<td>pub.{xsd,xml,std}</td>
<td>Third sample schema, instance document, and expected output respectively, after running xsdtest on them.</td>
</tr>
</tbody>
</table>

To build the sample programs, run `make`.

To build the programs and run them, comparing the actual output to expected output:

```
make sure
```
Error Handling in Transparent Mode

There is no notion of a valid event. Validity is the property of a document and not of the individual items and events of the document. The errors are:

- **XML_EVENT_FATAL_ERROR** - When the producer of XML events reports this error, the streaming validator returns this event back to the application and stops the validation process.

- **XML_EVENT_ERROR** - The streaming validator returns this event to the application when a validation error occurs. The application can then call XmlEvGetError() to get more information about the error.

If the application does not receive any **XML_EVENT_ERROR** or **XML_EVENT_FATAL_ERROR** events, the document is valid. Therefore, the application must handle these events and not ignore them.

These errors are not cached and the associated information is not available for later retrieval.

Streaming Validator Example

Here is an example in transparent mode:

```c
Example 20–1  Streaming Validator in Transparent Mode

#include "xmlev.h"
...
xmlevctx *ppevctx, *svevctx;
xmlctx *xctx
xsdctx *sctx;
if (!(xctx = XmlCreate(&xerr, (oratext *) "test")))
    printf("Failed to create XML context, error %u\n",
            (unsigned) xerr);
    ...
if (!(sctx = XmlSchemaCreate(xctx, &xerr, NULL)))
    printf("Failed to create schema context, error %u\n",
            (unsigned) xerr);
    ...
if (xerr = XmlSchemaLoad(sctx, "my_schema.xsd", NULL))
    printf("Failed to load schema, error %u\n",
            (unsigned) xerr);
    if(!(ppevctx = XmlEvCreatePPCtx(xctx, &xerr, NULL)))
        printf("Failed to create EVENT context, error %u\n",
                (unsigned) xerr);
    if(xerr = XmlEvLoadPPDoc(xctx, ppevctx, "file", "test.xml", 0, NULL))
        printf("Failed to load Document, error %u\n",
                (unsigned) xerr);
    ...
if(!(svevctx = XmlEvCreateSVCtx(xctx, sctx, ppevctx, &xerr)))
    printf("Failed to create SVcontext, error %u\n",
            (unsigned) xerr);
    ...
for(;;)
{
    xmlevtype cur_event;
    cur_event = XmlEvNext(svevctx);
}
```
Using Opaque Mode

In opaque mode, the streaming validator reads the instance document to be validated as a sequence of events from the producer, but does not pass the events to the application (consumer). It returns XMLERR_OK on success and an error number on failure.

After the schema has been loaded and the XML Events context has been initialized, the application can validate the document in this mode with a call to XmlEvSchemaValidate(). The signature of this function will take a pointer to the events context. The declaration is as follows:

```c
xmlerr XmlEvSchemaValidate(xmlctx *xctx, xsdctx *sctx, xmlevctx *evctx,
                             oratext **errmsg);
/* Returns (xmlerr), the error code */
```

Error Handling in Opaque Mode

When the streaming validator encounters an error, XmlEvSchemaValidate() returns an error number. This could be because of a parse error or a validation error. The application can then use the existing XmlEvGetError APIs to get the error message. The error message is parameterized and typically has all the errors leading up to the point where the streaming validator terminated.

Example of Opaque Mode Application

Here is an example in opaque mode:

```c
#include "xmlev.h"
...
xmlevctx *ppevtcx;
xmlctx   *xctx;
xsdctx   *sctx;
```
oratext **errmsg;
xmlerr xerr;

if (!(xctx = XmlCreate(&xerr, (oratext *) "test"))
   printf("Failed to create XML context, error \u
   , (unsigned) xerr);
...
if (!(sctx = XmlSchemaCreate(xctx, &xerr, NULL)))
   printf("Failed to create schema context, error \u
   , (unsigned) xerr);

...
if (xerr = XmlSchemaLoad(sctx, "my_schema.xsd", NULL))
   printf("Failed to load schema, error \u
   , (unsigned) xerr);
if (!(ppevctx = XmlEvCreatePPCtx(xctx, &xerr, NULL)))
   printf("Failed to create EVENT context, error \u
   , (unsigned) xerr);
if (xerr = XmlEvLoadPPDoc(xctx, ppevctx, "file", "test.xml", 0, NULL))
   printf("Failed to load Document, error \u
   , (unsigned) xerr);
if ((xerr = XmlEvSchemaValidate(xctx, sctx, ppevctx, errmsg)))
    {  
         printf("Validation Failed, Error: %s\n", errmsg);  
    }
...
XmlSchemaDestroy(sctx);
XmlEvDestroyCtx(ppevctx);
XmlDestroyCtx(xctx);

Enhancement of the Existing XmlSchemaLoad() Function

XmlSchemaLoad() was enhanced in 11g Release 1 (11.1) to work with an existing DOM. Previously, this function took two fixed arguments and a set of variable properties. The first argument is the schema context; the second is the URL location of the schema document. A new property was added to the set of variable arguments to provide access to the schema DOM given a URL. The property schema_dom_callback is a callback function provided by the application. If supplied, the schema load function will use this callback to access the DOM for the main schema as well as any included, imported, or redefined schemas. The callback signature is as follows:

typedef xmldocnode* (*xmlsch_dom_callback) (xmlctx *xctx, oratext *uri,
xmlerr *xerr);

This callback accepts a URI (the schema load function will pass in the URI of the document desired) and returns the document node. An example follows:

Example 20–3  XmlSchemaLoad() Example

#include "xmlev.h"
...
xmctxt *xctx;
xsdctx *sctx;
xmldocnode *doc;

if(!(xctx = XmlCreate(&xerr, (oratext *) "test"))
   printf("Failed to create XML context, error %u\n", (unsigned) xerr);
...
if (!(sctx = XmlSchemaCreate(xctx, &xerr, NULL)))
   printf("Failed to create schema context, error %u\n", (unsigned) xerr);
...
if(xerr = XmlSchemaLoad(sctx, schema_uri, "schema_dom_callback", func1, NULL))
Validation Options

You can supply options to the validation process using
XmlSchemaSetValidateOptions(). For example:

XmlSchemaSetValidateOptions(sctx, "ignore_id_constraint", (boolean)TRUE, NULL);

The options are:

- ignore_id_constraint (existing before 11.1)
- ignore_sch_location (existing before 11.1)
- ignore_par_val_rest (existing before 11.1)
- ignore_pattern_check: When this property is TRUE, the streaming validator ignores pattern-facet checks. The default is FALSE.
- no_events_for_defaults: When this property is TRUE, the streaming validator will not return events for default values added to the instance document. This option can be used in the transparent case only.

Example 20–4  Example of Streaming Validator Using New Options

```c
#include "xmlev.h"
...
levctx *ppevctx;
xmlctx   *xctx;
xsdctx   *sctx;
xmlerr   xerr;
oratext **errmsg;

if (!(xctx = XmlCreate(&xerr, (oratext *) "test"))
  printf("Failed to create XML context, error %u\n", (unsigned) xerr);
...
if (!(sctx = XmlSchemaCreate(xctx, &xerr, NULL)))
  printf("Failed to create schema context, error %u\n", (unsigned) xerr);
... If (xerr = XmlSchemaLoad(sctx, "my_schema.xsd", NULL))
  printf("Failed to load schema, error %u\n", (unsigned) xerr);
if(!(ppevctx = XmlEvCreatePPCtx(xctx, &xerr, "file", "test.xml", NULL))
  printf("Failed to create EVENT context, error %u\n", (unsigned) xerr);

if(xerr = XmlEvLoadPPDoc(xctx, ppevctx, "file", "test.xml", 0, NULL))
  printf("Failed to load Document, error %u\n", (unsigned) xerr);

XmlSchemaSetValidateOptions(sctx, "ignore_id_constraint", TRUE,
  "ignore_pattern_facet", TRUE, NULL);
if((xerr = XmlEvSchemaValidate(xctx,sctx, ppevctx, errmsg)))
  { printf("Validation Failed, Error: %s\n", errmsg);
  }
...
XmlSchemaDestroy(sctx);
XmlEvDestroyCtx(ppevctx);
XmlDestroyCtx(xctx);
```
Determining XML Differences Using C

The Oracle XDK includes components that help you to determine the differences between the contents of two XML documents and then to apply the differences (patch) to one of the XML documents.

This chapter contains these topics:

- Overview of XMLDiff in C
- Using XmlDiff
- Using XmlPatch
- Using XmlHash

Overview of XMLDiff in C

You can use Oracle XmlDiff to determine the differences between two similar XML documents. XmlDiff generates an Xdiff instance document that indicates the differences. The Xdiff instance document is an XML document that conforms to an XML schema, the Xdiff schema.

You can then use XmlPatch, which takes the Xdiff instance document and applies the changes to other documents. This can be used to apply the same changes to a large number of XML documents.

XmlDiff only supports the DOM API for input and output.

XmlPatch also supports the DOM for the input and patch documents.

XmlDiff and XmlPatch can be used through a C API or a command line tool, and they are exposed by two SQL functions.

An XmlHash C API is provided to compute the hash value of an XML tree or subtree. If hash values of two trees or subtrees are equal, the trees are identical to a very high probability.

Flow of Process for XMLDiff

The flow of the process is as follows:

1. The two input documents are compared by XmlDiff.
2. XmlDiff creates a Xdiff instance document.
3. The application can pass the Xdiff instance document to XmlPatch, if this is required.
4. XmlPatch can apply the differences captured from the comparison to other documents as specified by the application.

**Using XmlDiff**

XmlDiff compares the trees that represent the two input documents to determine differences.

Both input documents must use the same character-set encoding. The Xdiff (output) instance document has the same encoding as the data encoding (DOM encoding) of the input documents.

**User Options for Optimization**

There are two options for the comparison, known as optimizations:

- **Global Optimization - Default**
  
  The whole document trees are compared.

- **Local Optimization**
  
  Comparison is at the sibling level. Local optimization compares siblings under the corresponding parents from two trees.

Global optimization can take more time and space for large documents but always produces the smallest set of differences (the optimal difference). Local optimization is much faster, but may not produce the optimal difference.

**User Option for Hashing**

Hashing generally speeds up global optimization with a small possible loss in quality. Hashing improves the quality of the difference output, with local optimization. Using different hash levels may generate both local and global differences.

You can specify the use of hashing for both local and global optimization.

To specify hashing, provide the `hashLevel` parameter. If `hashLevel` is greater than 1, then only the DOMHash values are used for comparing all subtrees at depth \( \geq \) `hashLevel` of difference. If the hash values are equal, then the subtrees are presumed to be equal.

**How XmlDiff Looks at Input Documents**

XmlDiff ignores differences in the order of attributes while doing the comparison.

XmlDiff ignores DocType declarations. Files are not validated against the DTD.

XmlDiff ignores any differences in the namespace prefixes as long as the namespace prefixes refer to the same namespace URI. Otherwise, if two nodes have the same local name and content but differ in namespace URI, these differences are indicated.

---

**Note:** XmlDiff operates on its input documents in a nonschema-based way. It does not operate on elements or attributes in a type-aware manner.

---

**Using the XmlDiff Command Line Utility**

Table 21–1 describes command line options:
Sample Input Document

Example 21–1 is a sample xml document that can be used to explain updates resulting from using both XmlDiff and XmlPatch. It is followed by some hypothetical changes.

Example 21–1  book1.xml

```xml
<?xml version="1.0"?>
<booklist xmlns="http://booklist.oracle.com">
  <book>
    <title>Twelve Red Herrings</title>
    <author>Jeffrey Archer</author>
    <publisher>Harper Collins</publisher>
    <price>7.99</price>
  </book>
  <book>
    <title language="English">The Eleventh Commandment</title>
    <author>Jeffrey Archer</author>
    <publisher>McGraw Hill</publisher>
    <price>3.99</price>
  </book>
  <book>
    <title language="English" country="USA">C++ Primer</title>
    <author>Lippmann</author>
    <publisher>Harper Collins</publisher>
    <price>4.99</price>
  </book>
  <book>
    <title>Emperor's New Mind</title>
    <author>Roger Penrose</author>
    <publisher>Oxford Publishing Company</publisher>
    <price>15.9</price>
  </book>
  <book>
    <title>Evening News</title>
    <author>Arthur Hailey</author>
    <publisher>MacMillan Publishers</publisher>
    <price>9.99</price>
  </book>
</booklist>
```
Assume that there is another file, book2.xml, that looks just like the Example 21–1, "book1.xml" except that it results in the following:

Deletes "The Eleventh Commandment", a delete-node operation.

Changes the country code for the "C++ Primer" to US from USA, an update-node operation.

Adds a description to "Emperor's New Mind", an append-node operation.

Add the edition to "Evening News", an insert-node-before operation.

Updates the price of "Evening News", an update-node operation.

Sample Xdiff Instance Document

This section shows the Xdiff instance document produced by the comparison of these two XML files described in the previous section. The sections that follow explain the XML processing instructions and the operations on this document.

You can invoke XmlDiff as follows:

```
> xmldiff book1.xml book2.xml
```

You can also examine the sample application for arguments and flags.

Example 21–2 Sample Xdiff Instance Document

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xd:xdiff xsi:schemaLocation="http://xmlns.oracle.com/xdb/xdiff.xsd
xmlns:xd="http://xmlns.oracle.com/xdb/xdiff.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:oraxdfns_0="http://booklist.oracle.com">
  <?oracle-xmldiff operations-in-docorder="true" output-model="snapshot"
    diff-algorithm="global"?>
  <xd:delete-node xd:node-type="element" xd:xpath="/oraxdfns_0
    :booklist[1]/oraxdfns_0:book[2]/"/>
  <xd:update-node xd:node-type="attribute"
xd:parent-xpath="/oraxdfns_0:booklist[1]/oraxdfns_0:book[3]/oraxdfns_0
    :title[1]" xd:attr-local="country">
    <xd:content>US</xd:content>
  </xd:update-node>
  <xd:append-node xd:node-type="element" xd:parent-xpath="/oraxdfns_0
    :booklist[1]/oraxdfns_0:book[4]/">
    <xd:content>
      <oraxdfns_0:description>This is a classic</oraxdfns_0:description>
    </xd:content>
  </xd:append-node>
  <xd:insert-node-before xd:node-type="element" xd:xpath="/oraxdfns_0
    :booklist[1]/oraxdfns_0:book[5]/oraxdfns_0:author[1]/">
    <xd:content>
      <oraxdfns_0:edition>Hardcover</oraxdfns_0:edition>
    </xd:content>
  </xd:insert-node-before>
  <xd:update-node xd:node-type="text" xd:xpath="/oraxdfns_0
    :booklist[1]/oraxdfns_0:book[5]/oraxdfns_0:price[1]/text()[1]">
    <xd:content>12.99</xd:content>
  </xd:update-node>
</xd:xdiff>
```
Output Model and XML Processing Instructions

The Xdiff instance document uses some XML processing instructions (shown in bold in the previous section) that are used to represent certain aspects of the differencing process. See "Xdiff Schema" on page 21-6. These instructions and related options are:

- operations-in-docorder: Options are true or false:
  - true - The Xdiff instance document refers to the nodes from the first document in the same order as in the document.
  - false - The Xdiff instance document does not refer to the nodes from the first document in the same order as in the document.

The output of global optimization meets the operations-in-docorder requirement, but local optimization does not.

- output-model: Options are:
  - snapshot - XmlDiff generates output in snapshot model and follows the UNIX diff model. Each operation uses XPath as if no operations have been applied to the input document. This is the default. XmlPatch can only handle this model if operations-in-docorder is set to true and the XPaths are simple. Simple XPaths require a child axis, no wild cards, and must use positional predicates, such as /root[1]/child[2]/text()[2].
  - current - Each operation uses XPath as if all operations up to the previous one have been applied to the input document. Even though XmlDiff does not generate differences in the current model, XmlPatch can handle a hand-crafted diff document in the current model.

- diff-algorithm: Options indicate which optimization generated the differences.
  - Global optimization
  - Local optimization

See Also: "User Options for Optimization" on page 21-2

Xdiff Operations

XmlDiff captures differences using operations indicated by the Xdiff instance document.

Note the following about Xdiff operations:

- The parent-xpath attribute or xpath attribute specifies the XPath location of the parent node of the node to be operated on or XPath location of node.
- The node-type attribute specifies the type of the node to be operated on.
- The content child element specifies the new subtree or value appended or inserted.

The Xdiff operations, presented in the Xdiff Instance Document, are as follows:

- append-node:
  The append-node element specifies that a node of the given type is added as the last child of the given parent.

- insert-node-before:
  The insert-node-before element specifies that a node of the given type is inserted before the given reference node.
Using XmlDiff

- delete-node:

  The delete-node element specifies that the node be deleted along with all its children. This can be used to delete elements, comments, and so on.

- update-node:

  update-node specifies that the value associated with the node with the given XPath expression is updated to the new value, which is specified. Content is the value for a text node. The value of an attribute is the value for an attribute node.

  - Update for Text Nodes:

    * Generation of update node operations can be turned off by the user.
    * The value of an attribute is the value for an attribute node.
    * update-node is generated for text nodes only by global optimization.

  - Update for Elements:

    * XmlDiff does not generate update operations for element nodes.

      You can either manually modify the Xdiff instance document to create an update operation that works with XmlPatch, or provide a totally hand-written Xdiff instance document. All children of the element operated on by the update are deleted. Any new subtree specified under the content node is imported.

Format of Xdiff Instance Document

The output of XmlDiff, the Xdiff instance document, is in XML format and conforms to the Xdiff schema shown in the next section.

The output document contains a sequence of operations describing the differences between the two input documents. If you apply the differences from the first document, you get the second document.

Xdiff Schema

Example 21–3 shows the Xdiff schema to which the Xdiff instance document (output) adheres.

Example 21–3  Xdiff Schema: xdiff.xsd

```xml
<schema targetNamespace="http://xmlns.oracle.com/xdb/xdiff.xsd"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:xd="http://xmlns.oracle.com/xdb/xdiff.xsd"
version="1.0" elementFormDefault="qualified"
attributeFormDefault="qualified">
<annotation>
  <documentation xml:lang="en">
    Defines the structure of XML documents that capture the difference between two XML documents. Changes that are not supported by Oracle XmlDiff may not be expressible in this schema.

    'oracle-xmldiff' PI in Xdiff document:

    We use 'oracle-xmldiff' PI to describe certain aspects of the diff. The PI denotes values for 'operations-in-docorder' and 'output-model'. The output of XmlDiff has the PI always. If the user hand-codes a diff doc then it must also have the PI in it as the first child of top level xdiff element, to be able to call XmlPatch.
  </documentation>
</annotation>
```

21-6  Oracle XML Developer's Kit Programmer's Guide
operations-in-docorder:
Can be either 'true' or 'false'.
If true, the operations in the diff document refer to the
elements of the input doc in the same order as document order. Output of
global algorithm meets this requirement while local does not.

output-model:
output models for representing the diff. Can be either 'Snapshot' or
'Current'.

Snapshot model:
Each operation uses Xpaths as if no operations
have been applied to the input document. (like UNIX diff)
This is the model used in the output of XmlDiff. XmlPatch works with
this (and the current model too).
For XmlPatch to handle this model, "operations-in-docorder" must be
true and the Xpaths must be simple. (see XmlDiff C API documentation).

Current model:
Each operation uses Xpaths as if all operations till the previous one
have been applied to the input document. Works with XmlPatch even if
the 'operations-in-docorder' criterion is not met and the xpaths are
not simple.
<!-- Example:
<?oracle-xmldiff operations-in-docorder="true" output-model=
'snapshot' diff-algorithms="global"?&gt;
-->
</documentation>
</annotation>
<!-- Enumerate the supported node types -->
<simpleType name="xdiff-nodetype">
<restriction base="string">
<enumeration value="element"/>
<enumeration value="attribute"/>
<enumeration value="text"/>
<enumeration value="cdata"/>
<enumeration value="entity-reference"/>
<enumeration value="entity"/>
<enumeration value="processing-instruction"/>
<enumeration value="notation"/>
<enumeration value="comment"/>
</restriction>
</simpleType>

<element name="xdiff">
<complexType>
<choice minOccurs="0" maxOccurs="unbounded">
<element name="append-node">
<complexType>
<sequence>
<element name="content" type="anyType"/>
</sequence>
<attribute name="node-type" type="xd:xdiff-nodetype"/>
<attribute name="xpath" type="string"/>
<attribute name="parent-xpath" type="string"/>
<attribute name="attr-local" type="string"/>
<attribute name="attr-nsuri" type="string"/>
</complexType>
</element>
</choice>
</complexType>
</element>
Using XmlDiff in an Application

In an application, XmlDiff takes the source types and locations of the input documents as arguments. The source type can be a URL, file, orastream and stream context pointers, buffer, and buffer_length pointers or the pointer to a DOM document element (docelement).

XmlDiff returns the document node for the DOM for the Xdiff instance document.

XmlDiff builds the DOM for the two documents, if they are not already provided as DOM, before performing a comparison.
Using XmlDiff

Example 21–4 XMLDiff Application

```c
#include <xmldf.h>
...
xmlctx *xctx;
xmldocnode *doc1, *doc2, *doc3;
ulong hash_level;
oratex *s, *inp1 = "book1.xml", *inp2="book2.xml";
xmlerr err;
ub4 flags;
flags = 0; /* defaults : global algorithm */
hash_level = 0; /* no hashing */
/* create XML meta context */
if (!xctx = XmlCreate(&err, (oratext *) "XmlDiff", NULL))
{
  printf("Failed to create XML context, error %u\n", (unsigned) err);
  err_exit("Exiting");
}
/* Load the two input files */
if (!!(doc1 = XmlLoadDom(xctx, &err, "file", inp1, "discard_whitespace", TRUE,
       NULL)))
{
  printf("Parsing first file failed, error %u\n", (unsigned)err);
  err_exit((oratext *)"Exiting.");
}
if (!!(doc2 = XmlLoadDom(xctx, &err, "file", inp2, "discard_whitespace", TRUE,
       NULL)))
{
  printf("Parsing second file failed, error %u\n", (unsigned)err);
  err_exit((oratext *)"Exiting.");
}
/* run XmlDiff on the DOM trees. */
doc3 = XmlDiff(xctx, &err, flags, XMLDF_SRCT_DOM, doc1, NULL, XMLDF_SRCT_DOM,
       doc2, NULL,hash_level, NULL);
if(!doc3)
  printf("XmlDiff Failed, error %u\n", (unsigned)err);
else
{
  if(err != XMLERR_OK)
  printf("XmlDiff returned error %u\n", (unsigned)err);
  /* Now we have the DOM tree in doc3 which represent the Diff */
...
}
XmlFreeDocument(xctx, doc1);
XmlFreeDocument(xctx, doc2);
XmlFreeDocument(xctx, doc3);
XmlDestroy(xctx);
```

See Also: Oracle Database XML C API Reference, for the C API for the
flags that control the behavior of XmlDiff
Customized Output

A customized output builder stores differences in any format suitable to the application. You can create your own customized output builder, rather than using the default Xdiff instance document, generated by XmlDiff and which conforms to the Xdiff schema.

To do this, you must provide a callback that can be called after XmlDiff determines the differences. The differences are passed to the callback as an array of xmldfop. The callback may be called multiple times as the differences are being generated.

Using a customized output builder may perform better than using the default, because it does not have to maintain the internal state necessary for XPath generation.

By default, XmlDiff captures the differences in XML conforming to Xdiff schema. If necessary, plug in your own output builder. The differences are represented as an array xmldfop. You must write an output builder callback function. The function signature is:

```
xmlerr(*xdfobcb)(void *uctx, xmldfop *escript, ub4 escript_siz);
```

uctx is the user specific context.
escript is the array of size escript_siz:
diff[escript_siz]
mctx is the memory context.

Supply this memory context through properties to XmlDiff(). Use this memory context to allocate escript. You must later free escript.

Invoke the output builder callback after the differences have been found which happens even before the call to XmlDiff() returns. The output builder callback can be called multiple times.

**Example 21–5  Customized XMLDiff Output**

```
/* Sample usage: */
...
#include <orastruc.h>  // for 'oraprop' *
...
static oraprop diff_props[] = {
  ORAPROP(XMLDF_PROPN_CUSTOM_OB, XMLDF_PROPI_CUSTOM_OB, POINTER),
  ORAPROP(XMLDF_PROPN_CUSTOM_OBMCX, XMLDF_PROPI_CUSTOM_OBMCX, POINTER),
  ORAPROP(XMLDF_PROPN_CUSTOM_OBUCE, XMLDF_PROPI_CUSTOM_OBUCE, POINTER),
  { NULL }
};
...
oramemctx *mymemctx;
...
xmlerr myob(void *uctx, xmldfop *escript, ub4 escript_siz)
{
  /* process diff which is available in escript */

  /* free escript - the caller has to do this */
  OraMemFree(mymemctx, escript);
}

main()
{
  ...
```
Using XmlPatch

XmlPatch takes the Xdiff instance document, either as generated by XmlDiff or created by another mechanism, and follows the instructions in the Xdiff instance document to modify other XML documents as specified.

- Using the XmlPatch Command Line Utility
- Using XmlPatch in an Application

Using the XmlPatch Command Line Utility

Table 21–2 describes the XmlPatch command line options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e encoding</td>
<td>Specify default input-file encoding. If no encoding is specified in XML file, this encoding is assumed for input.</td>
</tr>
<tr>
<td>-E encoding</td>
<td>Specify output/data encoding. DOMs and patched document are created in this encoding. Default is UTF8.</td>
</tr>
<tr>
<td>-i</td>
<td>Interpret file names as URLs.</td>
</tr>
<tr>
<td>-h</td>
<td>Show this usage help.</td>
</tr>
</tbody>
</table>

Using XmlPatch in an Application

XmlPatch takes the source types and locations of the input document and the diff document as arguments. The source type can be a URL, file, orastream and stream context pointers, buffer and buffer_length pointers, or the pointer to a DOM document element (docelement).

See Also: Oracle Database XML C API Reference, for the C API for the flags that control the behavior of XmlPatch.

The modes that were set by the Xdiff schema affect how XmlPatch works.

If the output-model is Snapshot, XmlPatch only works if operations-in-docorder is TRUE.

If the output-model is Current, it is not necessary that operations-in-docorder be set to TRUE.

Example 21–6  Sample Application for XmlPatch

```c
#include <xmldf.h>
...
xmlctx     *xctx;
xmldocnode *doc1, *doc2;
```
oratext *s;
oratext *inpl = "book1.xml"; /* input document */
oratext *inp2 = "diff.xml", /* diff document */
xmlerr err;

/* create XML meta context */
if (!((xctx = XmlCreate(&err, (oratext *) "XmlPatch", NULL)))
{
    printf("Failed to create XML context, error %u\n",
    (unsigned) err);
    err_exit("Exiting");
}
/* Load the two input files */
if (!((doc1 = XmlLoadDom(xctx, &err, "file", inpl, "discard_whitespace", TRUE,
    NULL)))
{
    printf("Parsing first file failed, error %u\n", (unsigned)err);
    err_exit((oratext *)"Exiting.");
}
if (!((doc2 = XmlLoadDom(xctx, &err, "file", inp2, "discard_whitespace", TRUE,
    NULL)))
{
    printf("Parsing second file failed, error %u\n", (unsigned)err);
    err_exit((oratext *)"Exiting.");
}
/* call XmlPatch */
if (!XmlPatch(xctx, &err, 0, XMLDF_SRCT_DOM, doc1, NULL, XMLDF_SRCT_DOM,
    doc2, NULL, NULL))
{
    printf("XmlPatch Failed, error %u\n", (unsigned)err);
    XmlFreeDocument(xctx, doc1);
    XmlFreeDocument(xctx, doc2);
    XmlDestroy(xctx);
}

Using XmlHash

Oracle XDK provides XmlHash, which computes a hash value for an XML tree or subtree. If the hash values of two subtrees are equal, it is highly probable that they are the same XML. This can be used to do a quick comparison, for example, if you want to see if the XML tree is already in the database.

You can run XmlDiff again, if necessary, on any matches, to be absolutely certain there is a match. You can compute the hash value of the new document and query the database for it.

Example 21–7 shows a sample program that uses XmlHash.

**Example 21–7  XmlHash Program**
sword main(sword argc, char *argv[])
{

Using XmlHash
Oracle XDK provides XmlHash, which computes a hash value for an XML tree or subtree. If the hash values of two subtrees are equal, it is highly probable that they are the same XML. This can be used to do a quick comparison, for example, if you want to see if the XML tree is already in the database.

You can run XmlDiff again, if necessary, on any matches, to be absolutely certain there is a match. You can compute the hash value of the new document and query the database for it.

Example 21–7 shows a sample program that uses XmlHash.

**Example 21–7  XmlHash Program**
sword main(sword argc, char *argv[])
{
Using XmlHash

```c
xmlctx *xctx;
xmldfsrct srct;
oratext *data_encoding, *input_encoding, *s, *inp1;
ub1 flags;
xmlerr err;
ub4 num_args;
xmlhasht digest;
flags = 0; /* defaults */
srct = XMLDF_SRCT_FILE;
inp1 = "somexml.xml";
xctx = XmlCreate(&err, (oratext *) "XmlHash", NULL);

if (!xctx)
{
    /* handle error with creating xml context and exit */
    ...
}

/* run XmlHash */
err = XmlHash(xctx, &digest, 0, srct, inp1, NULL, NULL);
if(err)
    printf("XmlHash returned error:%d \n", err);
else
    txdfha_pd(digest);

XmlDestroy(xctx);

return (sword )err;
}

/* print bytes in xml hash */
static void txdfha_pd(xmlhasht digest)
{
    ub4 i;

    for(i = 0; i < digest.l_xmlhasht; i++)
        printf("%x ", digest.d_xmlhasht[i]);

    printf("\n");
}
```

Calling XmlDiff and XmlPatch

XmlDiff and XmlPatch can be called as command line tools and from the C language. They are also available as SQL functions.

See Also:

- Oracle Database SQL Language Reference, XMLDiff
- Oracle Database SQL Language Reference, XMLPatch
This chapter contains these topics:

- Introduction to SOAP for C
- SOAP C Functions
- SOAP Example 1: Sending an XML Document
- SOAP Example 2: A Response Asking for Clarification
- SOAP Example 3: Using POST

See Also: Oracle XML DB Developer’s Guide

Introduction to SOAP for C

The Simple Object Access Protocol (SOAP) is an XML protocol for exchanging structured and typed information between peers using HTTP and HTTPS in a distributed environment. Only HTTP 1.0 is supported in the XDK for 10g release 2. SOAP has three parts:

- The SOAP envelope which defines how to present what is in the message, who must process the message, and whether that processing is optional or mandatory.
- A set of serialization and deserialization rules for converting application data types to and from XML.
- A SOAP remote procedure call (RPC) that defines calls and responses.

Note: RPC and serialization/deserialization are not supported in this release.

 SOAP is operating system and language-independent because it is XML-based. This chapter presents the C implementation of the functions that read and write the SOAP message.

SOAP Version 1.2 is the definition of an XML-based message which is specified as an XML Infoset (an abstract data set, it could be XML 1.0) that gives a description of the message contents. Version 1.1 is also supported.
SOAP Messaging Overview

The Simple Object Access Protocol (SOAP) is a lightweight protocol for sending and receiving requests and responses across the Internet. Because it is based on XML and transport protocols such as HTTP, it is not blocked by most firewalls. SOAP is independent of operating system, implementation language, and object model.

The power of SOAP is its ability to act as the glue between heterogeneous software components. For example, Visual Basic clients can invoke CORBA services running on UNIX computers; Macintosh clients can invoke Perl objects running on Linux.

SOAP messages are divided into the following parts:

- An **envelope** that contains the message, defines how to process the message and who should process it, and whether processing is optional or mandatory. The Envelope element is required.
- A set of **encoding rules** that describe the datatypes for the application. These rules define a serialization mechanism that converts the application datatypes to and from XML.
- A **remote procedure call (RPC)** request and response convention. This required element is called a body element. The Body element contains a first subelement whose name is the name of a method. This method request element contains elements for each input and output parameter. The element names are the parameter names. RPC is not currently supported in this release.

SOAP is independent of any transport protocol. Nevertheless, SOAP used over HTTP for remote service invocation has emerged as a standard for delivering programmatic content over the Internet.

Besides being independent of transfer protocol, SOAP is also independent of operating system. In other words, SOAP enables programs to communicate even when they are written in different languages and run on different operating systems.

SOAP Message Format

SOAP messages are of the following types:

- Requests for a service, including input parameters
- Responses from the requested service, including return value and output parameters
- Optional fault elements containing error codes and information

In a SOAP message, the **payload** contains the XML-encoded data. The payload contains no processing information. In contrast, the message header may contain processing information.

**SOAP Requests**

In SOAP requests, the XML payload contains several elements that include the following:

- Root element
■ Method element

■ Header elements (optional)

Example 22–1 shows the format of a sample SOAP message request. A GetLastTradePrice SOAP request is sent to a StockQuote service. The request accepts a string parameter representing the company stock symbol and returns a float representing the stock price in the SOAP response.

**Example 22–1  SOAP Request Message**

POST /StockQuote HTTP/1.0
Host: www.stockquoteserver.com
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
SOAP-ENV:encodingStyle="http://www.w3.org/2003/05/soap-encoding/">
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>ORCL</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

In Example 22–1, the XML document is the SOAP message. The <SOAP-ENV:Envelope> element is the top-level element of the XML document. The payload is represented by the method element <m:GetLastTradePrice>. Note that XML namespaces distinguish SOAP identifiers from application-specific identifiers.

The first line of the header specifies that the request uses HTTP as the transport protocol:

POST /StockQuote HTTP/1.1

Because SOAP is independent of transport protocol, the rules governing XML payload format are independent of the use of HTTP for transport of the payload. This HTTP request points to the URI /StockQuote. Because the SOAP specification is silent on the issue of component activation, the code behind this URI determines how to activate the component and invoke the GetLastTradePrice method.

**Example of a SOAP Response  Example 22–2** shows the format of the response to the request in Example 22–1. The <Price> element contains the stock price for ORCL requested by the first message.

**Example 22–2  SOAP Response Message**

HTTP/1.0 200 OK
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
SOAP-ENV:encodingStyle="http://www.w3.org/2003/05/soap-encoding/">
  <SOAP-ENV:Body>
    <m:GetLastTradePriceResponse xmlns:m="Some-URI">
      <Price>13.5</Price>
    </m:GetLastTradePriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
The messages shown in Example 22–1 and Example 22–2 illustrate two-way SOAP messaging, that is, a SOAP request that is answered by a SOAP response. A one-way SOAP message does not require a SOAP message in response.

Using SOAP Clients

SOAP clients are user-written applications that generate XML documents. The documents make a request for a SOAP service and handle the SOAP response. The SOAP implementation in the XDK processes requests from any client that sends a valid SOAP request.

Note the following useful features of the SOAP client API:

- Supports a synchronous invocation model for requests and responses
- Facilitates the writing of client applications to make SOAP requests
- Encapsulates the creation of the SOAP request and the details of sending the request over the underlying transport protocol
- Supports a pluggable transport, allowing the client to easily change the transport (available transports include HTTP and HTTPS, but only HTTP 1.0 is supported in this release)

The SOAP client must perform the following steps to make a request and receive a response:

1. Gather all parameters that are needed to invoke a service.
2. Create a SOAP service request message, which is an XML message that is built according to the SOAP protocol. It contains all the values of all input parameters encoded in XML. This process is called serialization.
3. Submit the request to a SOAP server using a transport protocol that is supported by the SOAP server.
5. Determine the success or failure of the request by handling the SOAP Fault element.
6. Convert the returned parameter from XML to native datatype. This process is called deserialization.
7. Use the result as needed.

Using SOAP Servers

A SOAP server performs the following steps when executing a SOAP service request:

1. The SOAP server receives the service request.
2. The server parses the XML request and then decides whether to execute or reject the message.
3. If the message is executed, then the server determines whether the requested service exists.
4. The server converts all input parameters from XML into datatypes that the service understands.
5. The server invokes the service.
6. The server converts the return parameter to XML and generates a SOAP response message.
7. The server sends the response message back to the caller.

**SOAP C Functions**

The SOAP C implementation uses the xml.h header. A context of type xmlctx must be created before a SOAP context can be created.

HTTP aspects of SOAP are hidden from the user. SOAP endpoints are specified as a couple (binding, endpoint) where binding is of type xmlsoapbind and the endpoint is a (void *) depending on the binding. Currently, only one binding is supported, XMLSOAP_BIND_HTTP. For HTTP binding, the endpoint is an (OraText *) URL.

The SOAP layer creates and transports SOAP messages between endpoints, and decomposes received SOAP messages.

The C functions are declared in xmlsoap.h. Here is the beginning of that header file:

*Example 22–3  SOAP C Functions Defined in xmlsoap.h*

**FILE NAME**
xmlsoap.h - XML SOAP APIs

**FILE DESCRIPTION**
XML SOAP Public APIs

**PUBLIC FUNCTIONS**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmlSoapCreateCtx</td>
<td>Create and return a SOAP context</td>
</tr>
<tr>
<td>XmlSoapDestroyCtx</td>
<td>Destroy a SOAP context</td>
</tr>
<tr>
<td>XmlSoapCreateConnection</td>
<td>Create a SOAP connection object</td>
</tr>
<tr>
<td>XmlSoapDestroyConnection</td>
<td>Destroy a SOAP connection object</td>
</tr>
<tr>
<td>XmlSoapCall</td>
<td>Send a SOAP message &amp; wait for reply</td>
</tr>
<tr>
<td>XmlSoapCreateMsg</td>
<td>Create and return an empty SOAP message</td>
</tr>
<tr>
<td>XmlSoapDestroyMsg</td>
<td>Destroy a SOAP message created w/XmlSoapCreateMsg</td>
</tr>
<tr>
<td>XmlSoapGetEnvelope</td>
<td>Return a SOAP message's envelope</td>
</tr>
<tr>
<td>XmlSoapGetHeader</td>
<td>Return a SOAP message's envelope header</td>
</tr>
<tr>
<td>XmlSoapGetBody</td>
<td>Return a SOAP message's envelope body</td>
</tr>
<tr>
<td>XmlSoapAddHeaderElement</td>
<td>Adds an element to a SOAP header</td>
</tr>
<tr>
<td>XmlSoapGetHeaderElement</td>
<td>Gets an element from a SOAP header</td>
</tr>
<tr>
<td>XmlSoapAddBodyElement</td>
<td>Adds an element to a SOAP message body</td>
</tr>
<tr>
<td>XmlSoapGetBodyElement</td>
<td>Gets an element from a SOAP message body</td>
</tr>
<tr>
<td>XmlSoapSetMustUnderstand</td>
<td>Set mustUnderstand attr for SOAP hdr elem</td>
</tr>
<tr>
<td>XmlSoapGetMustUnderstand</td>
<td>Get mustUnderstand attr from SOAP hdr elem</td>
</tr>
<tr>
<td>XmlSoapSetRole</td>
<td>Set role for SOAP header element</td>
</tr>
<tr>
<td>XmlSoapGetRole</td>
<td>Get role from SOAP header element</td>
</tr>
<tr>
<td>XmlSoapSetRelay</td>
<td>Set relay Header element property</td>
</tr>
<tr>
<td>XmlSoapGetRelay</td>
<td>Get relay Header element property</td>
</tr>
<tr>
<td>XmlSoapSetFault</td>
<td>Set Fault in SOAP message</td>
</tr>
<tr>
<td>XmlSoapHasFault</td>
<td>Does SOAP message have a Fault?</td>
</tr>
<tr>
<td>XmlSoapGetFault</td>
<td>Return Fault code, reason, and details</td>
</tr>
</tbody>
</table>
SOAP Example 1: Sending an XML Document

Here is an XML document that illustrates a request to a travel company for a reservation on a plane flight from New York to Los Angeles for John Smith:

Example 22–4  Example 1 SOAP Message

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <m:reservation xmlns:m="http://travelcompany.example.org/reservation">
      ...
    </m:reservation>
  </env:Header>
</env:Envelope>
```

See Also: Oracle Database XML C API Reference for the C SOAP APIs
SOAP Example 1: Sending an XML Document

```xml
<env:Header
    env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
    env:mustUnderstand="true">
  <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
  <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
</m:reservation>
<n:passenger xmlns:n="http://mycompany.example.com/employees"
    env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
    env:mustUnderstand="true">
  <n:name>John Smith</n:name>
</n:passenger>
</env:Header>
<env:Body>
  <p:itinerary xmlns:p="http://travelcompany.example.org/reservation/travel">
    <p:departure>
      <p:departing>New York</p:departing>
      <p:arriving>Los Angeles</p:arriving>
      <p:departureDate>2001-12-14</p:departureDate>
      <p:departureTime>late afternoon</p:departureTime>
      <p:seatPreference>aisle</p:seatPreference>
    </p:departure>
    <p:return>
      <p:departing>Los Angeles</p:departing>
      <p:arriving>New York</p:arriving>
      <p:departureDate>2001-12-20</p:departureDate>
      <p:departureTime>mid-morning</p:departureTime>
      <p:seatPreference/>
    </p:return>
  </p:itinerary>
  <q:lodging xmlns:q="http://travelcompany.example.org/reservation/hotels">
    <q:preference>none</q:preference>
  </q:lodging>
</env:Body>
</env:Envelope>
```

The example used to create the XML document, send it, and receive and decompose a reply is simplified. There is some minimal error checking. The DEBUG option is shown for correcting anomalies. The program may not work on all operating systems. To send this XML document, the first client C program follows these steps:

- After declaring variables in `main()`, an XML context, `xctx`, is created using `XmlCreate()` and the context is then used to create a SOAP context, `ctx`, using `XmlSoapCreateCtx()`.
- To construct the message, `XmlSoapCreateMsg()` is called and returns an empty SOAP message.
- The header is constructed using `XmlSoapAddHeaderElement()`, `XmlSoapSetRole()`, `XmlSoapSetMustUnderstand()`, and `XmlDomAddTextElem()` to fill in the envelope with text.
- The body elements are created by `XmlSoapAddBodyElement()`, `XmlDomCreateElemNS()`, and a series of calls to `XmlDomAddTextElem()`. Then `XmlDomAppendChild()` completes the section of the body specifying the New York to Los Angeles flight.
- The return flight is built in an analogous way. The lodging is added with another `XmlSoapAddBodyElement()` call.
The connection must be created next with `XmlSoapCreateConnection()`, specifying HTTP binding (the only binding available now) and an endpoint URL.

The function `XmlSoapCall()` sends the message over the defined connection by means of the SOAP server, and then waits for the reply.

The message reply is returned in the form of another SOAP message. This is done with `XmlSaveDom()` and `XmlSoapHasFault()` used with `XmlSoapGetFault()` to check for a fault and analyze the fault. The fault is parsed into its parts, which is output in this example.

If there was no fault returned, this is followed by `XmlSoapGetBody()` to return the envelope body. `XmlSaveDom()` completes the analysis of the returned message.

To clean up, use `XmlSoapDestroyMsg()` on the message and on the reply, `XmlDestroyCtx()` to destroy the SOAP context, and `XmlDestroy()` to destroy the XML context.

The C client program for Example 1 is:

**Example 22–5  Example 1 SOAP C Client**

```c
#ifndef S_ORACLE
#include <s.h>
#endif

#ifndef XML_ORACLE
#include <xml.h>
#endif

#ifndef XMLSOAP_ORACLE
#include <xmlsoap.h>
#endif

#define MY_URL "http://my_url.com"

/* static function declaration */
static xmlerr add_ns_decl(xmlsoapctx *ctx, xmlctx *xctx, xmlelemnode *elem,
                           oratext *pfx, oratext *uri);

sb4 main( sword argc, char *argv[])
{
    xmlctx  *xctx;
    xmlerr  xerr;
    xmlsoapctx *ctx;
    oratext *url;
    xmlsoapcon *con;
    xmldocnode *msg1, *reply, *msg2, *msg3;
    xmlelemnode *body, *header;
    boolean has_fault;
    xmlelemnode *detail;
    oratext *comp_uri   = "http://travelcompany.example.org/";
    oratext *mres_uri   = "http://travelcompany.example.org/reservation";
    oratext *trav_uri   = "http://travelcompany.example.org/reservation/travel";
    oratext *hotel_uri  = "http://travelcompany.example.org/reservation/hotels";
```
oratext *npas_uri = "http://mycompany.example.com/employees";
oratext *tparty_uri = "http://thirdparty.example.org/transaction";
oratext *estyle_uri = "http://example.com/encoding";
oratext *soap_style_uri = "http://www.w3.org/2003/05/soap-encoding";
oratext *estyle = "env:encodingStyle";
oratext *finance_uri = "http://mycompany.example.com/financial";

if (!(xctx = XmlCreate(&xerr, (oratext *)"SOAP_test",NULL)))
{
    printf("Failed to create XML context, error %u\n", (unsigned) xerr);
    return EX_FAIL;
}
/* Create SOAP context */
if (!(ctx = XmlSoapCreateCtx(xctx, &xerr, (oratext *) "example", NULL)))
{
    printf("Failed to create SOAP context, error %u\n", (unsigned) xerr);
    return EX_FAIL;
}

/* EXAMPLE 1 */
/* construct message */
if (!(!(msg1 = XmlSoapCreateMsg(ctx, &xerr)))
{
    printf("Failed to create SOAP message, error %u\n", (unsigned) xerr);
    return xerr;
}
res = XmlSoapAddHeaderElement(ctx, msg1, "m:reservation", mres_uri, &xerr);
xerr = XmlSoapSetRole(ctx, res, XMLSOAP_ROLE_NEXT);
xerr = XmlSoapSetMustUnderstand(ctx, res, TRUE);
(void) XmlDomAddTextElem(xctx, res, mres_uri, "m:reference",
    "uuid:093a2dal-q345-739r-ba5d-pqff98fe8j7d");
(void) XmlDomAddTextElem(xctx, res, mres_uri, "m:dateAndTime",
    "2001-11-29T13:20:00.000-05:00");
pas = XmlSoapAddHeaderElement(ctx, msg1, "n:passenger", npas_uri, &xerr);
xerr = XmlSoapSetRole(ctx, pas, XMLSOAP_ROLE_NEXT);
xerr = XmlSoapSetMustUnderstand(ctx, pas, TRUE);
(void) XmlDomAddTextElem(xctx, pas, npas_uri, "n:name",
    "John Smith");
/* Fill body */
/* Itinerary */
itin = XmlSoapAddBodyElement(ctx, msg1, "p:itinerary", trav_uri, &xerr); /* Departure */
departure = XmlDomCreateElemNS(xctx, msg1, trav_uri, "p:departure");
(void) XmlDomAddTextElem(xctx, departure, trav_uri,
    "p:departing", "New York");
(void) XmlDomAddTextElem(xctx, departure, trav_uri,
    "p:arriving", "Los Angeles");
(void) XmlDomAddTextElem(xctx, departure, trav_uri,
    "p:departureDate", "2001-12-14");
(void) XmlDomAddTextElem(xctx, departure, trav_uri,
    "p:departureTime", "late afternoon");
(void) XmlDomAddTextElem(xctx, departure, trav_uri,
    "p:seatPreference", "aisle");
XmlDomAppendChild(xctx, itin, departure);
/* Return */
ret = XmlDomCreateElemNS(xctx, msg1, trav_uri, "p:return");
(void) XmlDomAddTextElem(xctx, ret, trav_uri,
        "p:departing", "Los Angeles");
(void) XmlDomAddTextElem(xctx, ret, trav_uri,
        "p:arriving", "New York");
(void) XmlDomAddTextElem(xctx, ret, trav_uri,
        "p:departureDate", "2001-12-20");
(void) XmlDomAddTextElem(xctx, ret, trav_uri,
        "p:departureTime", "mid-morning");
pref = XmlDomCreateElemNS(xctx, msg1, trav_uri, "p:seatPreference");
(void) XmlDomAppendChild(xctx, ret, pref);
XmlDomAppendChild(xctx, itin, ret);

/* Lodging */
lodging = XmlSoapAddBodyElement(ctx, msg1, "q:lodging", hotel_uri, &xerr);
(void) XmlDomAddTextElem(xctx, lodging, hotel_uri,
        "q:preference", "none");

#ifdef DEBUG
/* dump the message in debug mode */
printf("Message:\n");
XmlSaveDom(xctx, &xerr, msg1, "stdio", stdout, "indent_step", 1, NULL);
#endif

/* END OF EXAMPLE 1 */

/* create connection */
url = MY_URL;
if (!(con = XmlSoapCreateConnection(ctx, &xerr, XMLSOAP_BIND_HTTP,
        url, NULL, 0, NULL, 0,
        "XTest: baz", NULL)))
{
    printf("Failed to create SOAP connection, error %u\n", (unsigned) xerr);
    return xerr;
}
reply = XmlSoapCall(ctx, con, msg1, &xerr);
XmlSoapDestroyConnection(ctx, con);
if (!reply)
{
    printf("Call failed, no message returned.\n");
    return xerr;
}

#ifdef DEBUG
printf("Reply:\n");
XmlSaveDom(xctx, &xerr, reply, "stdio", stdout, NULL);
#endif

printf("\n==== Header:\n");
header = XmlSoapGetHeader(ctx, reply, &xerr);
if (!header)
{
    printf("NULL\n");
}
else
    XmlSaveDom(xctx, &xerr, header, "stdio", stdout, NULL);

/* check for fault */
has_fault = XmlSoapHasFault(ctx, reply, &xerr);
if(has_fault)
{
    lang = NULL;
xerr = XmlSoapGetFault(ctx, reply, &code, &reason, &lang,
                        &node, &role, &detail);
if (xerr)
{
    printf("error getting Fault %d\n", xerr);
    return EX_FAIL;
}
if(code)
    printf(" Code -- %s\n", code);
else
    printf(" NO Code\n");
if(reason)
    printf(" Reason -- %s\n", reason);
else
    printf(" NO Reason\n");
if(lang)
    printf(" Lang -- %s\n", lang);
else
    printf(" NO Lang\n");
if(node)
    printf(" Node -- %s\n", node);
else
    printf(" NO Node\n");
if(role)
    printf(" Role -- %s\n", role);
else
    printf(" NO Role\n");
if(detail)
{
    printf(" Detail\n");
    XmlSaveDom(xctx, &xerr, detail, "stdio", stdout, NULL);
    printf("\n");
}
else
    printf(" NO Detail\n");
}
else
{
    body = XmlSoapGetBody(ctx, reply, &xerr);
    printf("==== Body:\n ");
    if (!body)
{
    printf("NULL\n");
    return EX_FAIL;
}
    XmlSaveDom(xctx, &xerr, body, "stdio", stdout, NULL);
}
(void) XmlSoapDestroyMsg(ctx, reply);
(void) XmlSoapDestroyMsg(ctx, msg1);
(void) XmlSoapDestroyCtx(ctx);
XmlDestroy(xctx);
SOAP Example 2: A Response Asking for Clarification

The travel company wants to know which airport in New York the traveller, John Smith, will depart from. The choices are JFK for Kennedy, EWR for Newark, or LGA for LaGuardia. So the following reply is sent:

Example 22–6  Example 2 SOAP Message

<?xml version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
<env:Header>
<m:reservation xmlns:m="http://travelcompany.example.org/reservation" env:role="http://www.w3.org/2003/05/soap-envelope/role/next" env:mustUnderstand="true">
<n:passenger xmlns:n="http://mycompany.example.com/employees" env:role="http://www.w3.org/2003/05/soap-envelope/role/next" env:mustUnderstand="true">
<n:name>John Smith</n:name>
</n:passenger>
</env:Header>
<env:Body>
<p:itineraryClarification xmlns:p="http://travelcompany.example.org/reservation/travel">
<p:departure>
<p:departing>
<p:airportChoices>
JFK LGA EWR
</p:airportChoices>
</p:departing>
</p:departure>
<p:return>
<p:arriving>
<p:airportChoices>
JFK LGA EWR
</p:airportChoices>
</p:arriving>
</p:return>
</p:itineraryClarification>
</env:Body>
</env:Envelope>

To send this XML document as a SOAP message, substitute the following code block for the lines beginning with /* EXAMPLE 1 */ and ending with /* END OF EXAMPLE 1 */ in Example 22–5, "Example 1 SOAP C Client":

Example 22–7  Example 2 SOAP C Client

#define XMLSOAP_MAX_NAME     1024

/* we need this function for examples 2 and 3 */
static xmlerr add_ns_decl(xmlsoapctx  *ctx, xmlctx *xtctx, xmllemnode *elem,
    oratext *pxf, oratext *xuri)
{
    oratext     *aq, aqbuf[XMLSOAP_MAX_NAME];
xmldcnodex *doc;
oratext     *xmlns = 'xmlns:';
SOAP Example 2: A Response Asking for Clarification

/* if no room for "xmlns:usersprefix\0" then fail now */
if ((strlen((char *)pfx) + strlen((char *)xmlns)) > sizeof(aqbuf))
    return EX_FAIL;
(void) strcpy((char *)aqbuf, (char *)xmlns);
strcat((char *)aqbuf, (char *)pfx);
doc = XmlDomGetOwnerDocument(xctx, elem);
aq = XmlDomSaveString(xctx, doc, aqbuf);
XmlDomSetAttrNS(xctx, elem, uri, aq, uri);
return XMLERR_OK;
}

/* EXAMPLE 2 */
/* construct message */
if (!(msg2 = XmlSoapCreateMsg(ctx, &xerr))){
    printf("Failed to create SOAP message, error %u\n", (unsigned) xerr);
    return xerr;
}
res = XmlSoapAddHeaderElement(ctx, msg2, "m:reservation", mres_uri, &xerr);
xerr = XmlSoapSetRole(ctx, res, XMLSOAP_ROLE_NEXT);
xerr = XmlSoapSetMustUnderstand(ctx, res, TRUE);
(void) XmlDomAddTextElem(xctx, res, mres_uri, "m:reference",
    "uuid:093a2dal-q345-739r-ba5d-pqff98fe8j7d");
(void) XmlDomAddTextElem(xctx, res, mres_uri, "m:dateAndTime",
    "2001-11-29T13:35:00.000-05:00");
pas = XmlSoapAddHeaderElement(ctx, msg2, "n:passenger", npas_uri, &xerr);
xerr = XmlSoapSetRole(ctx, pas, XMLSOAP_ROLE_NEXT);
xerr = XmlSoapSetMustUnderstand(ctx, pas, TRUE);
(void) XmlDomAddTextElem(xctx, pas, npas_uri, "n:name",
    "John Smith");
/* Fill body */
/* Itinerary */
itin = XmlSoapAddBodyElement(ctx, msg2, "p:itineraryClarification",
    trav_uri, &xerr);
/* Departure */
departure = XmlDomCreateElemNS(xctx, msg2, trav_uri, "p:departure");
departing = XmlDomCreateElem(xctx, msg2, "p:departing");
(void) XmlDomAddTextElem(xctx, departing, trav_uri,
    "p:airportChoices", "JFK LGA EWR");
(void) XmlDomAppendChild(xctx, departure, departing);
XmlDomAppendChild(xctx, itin, departure);
/* Return */
ret = XmlDomCreateElemNS(xctx, msg2, trav_uri, "p:return");
arriving = XmlDomCreateElemNS(xctx, msg2, trav_uri, "p:arriving");
(void) XmlDomAddTextElem(xctx, arriving, trav_uri,
    "p:airportChoices", "JFK LGA EWR");
XmlDomAppendChild(xctx, ret, arriving);
XmlDomAppendChild(xctx, itin, ret);
#ifdef DEBUG
XmlSaveDom(xctx, &xerr, msg2, "stdio", stdout, "indent_step", 1, NULL);
#endif
### SOAP Example 3: Using POST

Credit card information for John Smith is sent in the final XML document using the POST method. The `XmlSoapCall()` writes the HTTP header that precedes the XML message in the following example:

#### Example 22–8  Example 3 SOAP Message

```
POST /Reservations HTTP/1.0
Host: travelcompany.example.org
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
  <env:Header>
    <t:transaction
      xmlns:t="http://thirdparty.example.org/transaction"
      env:encodingStyle="http://example.com/encoding"
      env:mustUnderstand="true" >5</t:transaction>
  </env:Header>
  <env:Body>
    <m:chargeReservation
      xmlns:m="http://travelcompany.example.org/">
      <m:reservation
        xmlns:m="http://travelcompany.example.org/reservation">
        <m:code>FT35ZBQ</m:code>
        <o:creditCard
          xmlns:o="http://mycompany.example.com/financial">
          <n:name
            xmlns:n="http://mycompany.example.com/employees">
            John Smith
          </n:name>
          <o:number>123456789099999</o:number>
          <o:expiration>2005-02</o:expiration>
        </o:creditCard>
      </m:reservation>
    </m:chargeReservation>
  </env:Body>
</env:Envelope>
```

The C Client includes the following code block which is substituted like the second example in Example 22–5, "Example 1 SOAP C Client":

#### Example 22–9  Example 3 SOAP C Client

```c
#define XMLSOAP_MAX_NAME 1024

/* we need this function for examples 2 and 3 */
static xmlerr add_ns_decl(xmlsoapctx *ctx, xmlctx *xctx, xmlelemnode *elem,
  oratext *pfx, oratext *uri)
{
  oratext *aq, aqbuf[XMLSOAP_MAX_NAME];
  xmldocnode *doc;
  oratext *xmlns = "xmlns:";

  /* if no room for 'xmlns:usersprefix\0' then fail now */
  if ((strlen((char *)pfx) + strlen((char *)xmlns)) >
    sizeof(aqbuf))
    return EX_FAIL;
  strcpy((char *)aqbuf, (char *)xmlns);
  strcat((char *)aqbuf, (char *)pfx);
```
SOAP Example 3: Using POST

Using SOAP with the C XDK

```c
DOC = XmlDomGetOwnerDocument(xctx, elem);
aq = XmlDomSaveString(xctx, doc, aqbuf);
XmlDomAttrNS(xctx, elem, uri, aq, uri);
return XMLERR_OK;
}
/* EXAMPLE 3 */
if (!(msg3 = XmlSoapCreateMsg(ctx, &xerr)))
{
    printf("Failed to create SOAP message, error %u\n", (unsigned) xerr);
    return xerr;
}
trans = XmlSoapAddHeaderElement(ctx, msg3, "t:transaction", tparty_uri, &xerr);
xerr = XmlSoapSetMustUnderstand(ctx, trans, TRUE);
XmlDomAttr(xctx, trans, estyle, estyle_uri);
text = XmlDomCreateText(xctx, msg3, "5");
XmlDomAppendChild(xctx, trans, text);
/* Fill body */
/* Charge Reservation */
charge = XmlSoapAddBodyElement(ctx, msg3, "m:chargeReservation", comp_uri, &xerr);
XmlDomAttr(xctx, charge, estyle, soap_style_uri);
res = XmlDomCreateElemNS(xctx, msg3, "m:reservation");
if (add_nsDecl(ctx, xctx, res, "m", mres_uri))
    return EX_FAIL;
(void) XmlDomAddTextElem(xctx, res, mres_uri,
"m:code", "FT35ZBQ");
(void) XmlDomAppendChild(xctx, charge, res);
/* create card elem with namespace */
card = XmlDomCreateElemNS(xctx, msg3, finance_uri, "o:creditCard");
if (add_nsDecl(ctx, xctx, card, "o", finance_uri))
    return EX_FAIL;
name = XmlDomAddTextElem(xctx, card, npas_uri,
"n:name", "John Smith");
/* add namespace */
if (add_nsDecl(ctx, xctx, name, "n", npas_uri))
    return EX_FAIL;
(void) XmlDomAddTextElem(xctx, card, finance_uri,
"o:number", "123456789099999");
(void) XmlDomAddTextElem(xctx, card, finance_uri,
"o:expiration", "2005-02");
(void) XmlDomAppendChild(xctx, charge, card);
#ifdef DEBUG
XmlSaveDom(xctx, &xerr, msg3, "stdio", stdout, "indent_step", 1, NULL);
#endif
```
Part III

Oracle XDK for C++

This part contains chapters that describe how the Oracle XDK is used for developing applications in C++.

This part contains the following chapters:

- Chapter 23, "Getting Started with C++ XDK Components"
- Chapter 24, "Overview of the Unified C++ Interfaces"
- Chapter 25, "Using the XML Parser for C++"
- Chapter 26, "Using the XSLT Processor for C++"
- Chapter 27, "Using the XML Schema Processor for C++"
- Chapter 28, "Using the XPath Processor for C++"
- Chapter 29, "Using the XML Class Generator for C++"
This chapter describes the Oracle Database installation of the XDK. Note that the C++
demo programs are located on the Examples media.

This chapter contains these topics:

- Installing the C++ XDK Components
- Configuring the UNIX Environment for C++ XDK Components
- Configuring the Windows Environment for C++ XDK Components

Installing the C++ XDK Components

The C++ XDK components are included with Oracle Database and Oracle Application
Server.

Refer to "Installing the XDK" on page 1-17 for installation instructions.

See Also: "Overview of Oracle XML Developer’s Kit (XDK)" on
page 1-1 for a list of the C++ XDK components

Configuring the UNIX Environment for C++ XDK Components

This section contains the following topics:

- C++ XDK Component Dependencies on UNIX
- Setting C++ XDK Environment Variables on UNIX
- Testing the C++ XDK Runtime Environment on UNIX
- Setting Up and Testing the C++ XDK Compile-Time Environment on UNIX
- Verifying the C++ XDK Component Version on UNIX

C++ XDK Component Dependencies on UNIX

The C++ libraries described in this section are located in $ORACLE_HOME/lib. The C and
C++ XDK components are contained in the library:

libxml11.a

In addition to the C XDK components described in "C XDK Component Dependencies
on UNIX" on page 16-2, the library includes the XML class generator, which creates
C++ source files based on an input DTD or XML Schema.
Table 16–1 in "C XDK Component Dependencies on UNIX" on page 16-2 describes the Oracle CORE and Globalization Support libraries on which the C XDK components (UNIX) depend. The library dependencies are the same for C and C++.

Setting C++ XDK Environment Variables on UNIX

Table 16–2 in "Setting C XDK Environment Variables on UNIX" on page 16-3 describes the UNIX environment variables required for use with the C XDK components. The environment variables are the same for C and C++.

Testing the C++ XDK Runtime Environment on UNIX

You can test your environment by running any of the utilities described in Table 16–3 in "Testing the C XDK Runtime Environment on UNIX" on page 16-3. These utilities are C utilities that do not have C++ versions.

Setting Up and Testing the C++ XDK Compile-Time Environment on UNIX

Both the C and C++ header files are located in $ORACLE_HOME/xdk/include. Table 23–1 describes the C++ header files. Table 16–4 in "Setting Up and Testing the C XDK Compile-Time Environment on UNIX" on page 16-4 describes the C header files. Your runtime environment must be set up before you can compile your C++ code.

Table 23–1  Header Files in the C++ XDK Compile-Time Environment

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oraxml.hpp</td>
<td>Includes the Oracle9i XML ORA datatypes and the public ORA APIs included in libxml.a (for backward compatibility only).</td>
</tr>
<tr>
<td>oraxmlcg.h</td>
<td>Includes the C APIs for the C++ class generator (for backward compatibility only).</td>
</tr>
<tr>
<td>oraxsd.hpp</td>
<td>Includes the Oracle9i XSD validator datatypes and APIs (for backward compatibility only)</td>
</tr>
<tr>
<td>xml.hpp</td>
<td>Handles the Unified DOM APIs transparently, whether you use them through OCI or standalone</td>
</tr>
<tr>
<td>xmlotn.hpp</td>
<td>Includes the common APIs, whether you compile standalone or use OCI and the Unified DOM</td>
</tr>
<tr>
<td>xmlctx.hpp</td>
<td>Includes the initialization and exception-handling public APIs</td>
</tr>
</tbody>
</table>

Testing the C++ XDK Compile-Time Environment on UNIX

The simplest way to test your compile-time environment is to run the make utility on the sample programs. The demo programs are located on the Examples media rather than the Oracle Database CD. After you install these programs, they will be located in $ORACLE_HOME/xdk/demo/cpp.

Build and run the sample programs by executing the following commands at the system prompt:

```
  cd $ORACLE_HOME/xdk/demo/cpp
  make
```

Verifying the C++ XDK Component Version on UNIX

To obtain the version of XDK you are using, change into $ORACLE_HOME/lib and run the following command as the system prompt:
Configuring the Windows Environment for C++ XDK Components

This section contains the following topics:

- C++ XDK Component Dependencies on Windows
- Setting C++ XDK Environment Variables on Windows
- Testing the C++ XDK Runtime Environment on Windows
- Setting Up and Testing the C++ XDK Compile-Time Environment on Windows
- Using the C++ XDK Components with Visual C/C++

C++ XDK Component Dependencies on Windows

The C++ libraries described in this section are located in %ORACLE_HOME%/lib. The XDK C and C++ components are contained in the following Windows library:

libxml10.dll

Table 16–5 in "C XDK Component Dependencies on Windows" on page 16-5 describes the Oracle CORE and Globalization Support libraries on which the C components for Windows depend. The library dependencies are the same for C and C++.

Setting C++ XDK Environment Variables on Windows

Table 16–6 in "Setting C XDK Environment Variables on Windows" on page 16-6 describes the Windows environment variables required for use with the XDK C components. The environment variables are the same for C and C++.

Testing the C++ XDK Runtime Environment on Windows

You can test your environment by running any of the utilities described in Table 16–7 in "Testing the C XDK Runtime Environment on Windows" on page 16-6. These utilities are C utilities that do not have C++ versions.

Setting Up and Testing the C++ XDK Compile-Time Environment on Windows

Table 23–1 in the section "Setting Up and Testing the C++ XDK Compile-Time Environment on UNIX" on page 23-2 describes the header files required for compilation of the C components on Windows. The relative filenames are the same on both UNIX and Windows installations.

On Windows the header files are located in %ORACLE_HOME%/xdk/include. Note that your runtime environment must be set up before you can compile your code.

Testing the C++ XDK Compile-Time Environment on Windows

You can test your compile-time environment by compiling the demo programs, which are located in %ORACLE_HOME%/xdk/demo/cpp if you have installed the Oracle Database Examples media.

The procedure for setting the C++ compiler path is identical to the procedure described in "Setting the C XDK Compiler Path on Windows" on page 16-8. The procedure for editing the Make.bat files is identical to the procedure described in "Editing the Make.bat Files on Windows" on page 16-7.
Using the C++ XDK Components with Visual C/C++

You can set up a project in Microsoft Visual C/C++ and use it for the demos included in the XDK. Refer to "Using the C XDK Components with Visual C/C++ on Windows" on page 16-8 for instructions.
Overview of the Unified C++ Interfaces

This chapter contains these topics:

- What is the Unified C++ API?
- Accessing the C++ Interface
- OracleXML Namespace
- Ctx Namespace
- IO Namespace
- Tools Package
- Error Message Files

What is the Unified C++ API?

Unified C++ APIs for XML tools represent a set of C++ interfaces for Oracle XML tools. This unified approach provides a generic, interface-based framework that allows XML tools to be improved, updated, replaced, or added without affecting any interface-based user code, and minimally affecting application drivers and, possibly, application configuration. All three kinds of C++ interfaces: abstract classes, templates, and implicit interfaces represented by generic template parameters, are used by the unified framework.

**Note:** Use the new unified C++ API in `xml.hpp` for new XDK applications. The old C++ API in `oraxml.hpp` is deprecated and supported only for backward compatibility, but will not be enhanced. It will be removed in a future release.

These C++ APIs support the W3C specification as closely as possible; however, Oracle cannot guarantee that the specification is fully supported by our implementation because the W3C specification does not cover C++ implementations.

Accessing the C++ Interface

The C++ interface is provided with Oracle Database. Sample files are located in `$ORACLE_HOME/xdk/demo/cpp`.

`readme.html` in the root directory of the software archive contains release specific information including bug fixes and API additions.
OracleXML Namespace

OracleXml is the C++ namespace for all XML C++ interfaces. It contains common interfaces and namespaces for different XDK packages. The following namespaces are included:

- **Ctx** - namespace for TCtx related declarations
- **Dom** - namespace for DOM related declarations
- **Parser** - namespace for parser and schema validator declarations
- **IO** - namespace for input and output source declarations
- **Xsl** - namespace for XSLT related declarations
- **XPath** - namespace for XPath related declarations
- **Tools** - namespace for Tools::Factory related declarations

OracleXml is fully defined in the file `xml.hpp`. Another namespace, `XmlCtxNS`, visible to users, is defined in `xmlctx.hpp`. That namespace contains C++ definitions of data structures corresponding to C level definitions of the `xmlctx` context and related data structures. While there is no need for users to know details of that namespace, `xmlctx.hpp` needs to be included in most application main modules.

Multiple encodings are currently supported on the base of the `oratext` type that is currently supposed to be used by all implementations. All strings are represented as `oratext*`.

OracleXML Interfaces

XMLException Interface - This is the root interface for all XML exceptions.

Ctx Namespace

The Ctx namespace contains data types and interfaces related to the TCtx interface.

OracleXML Datatypes

DATATYPE encoding - a particular supported encoding. The following kinds of encodings (or encoding names) are supported:

- `data_encoding`
- `default_input_encoding`
- `input_encoding` - overwrites the previous one
- `error_language` - gets overwritten by the language of the error handler, if specified
- `DATATYPE encodings` - array of encodings.

Ctx Interfaces

ErrorHandler Interface - This is the root error handler class. It deals with local processing of errors, mainly from the underlying C implementation. It may throw XmlException in some implementations. But this is not specified in its signature in order to accommodate needs of all implementations. However, it can create exception objects. The error handler is passed to the `TCtx` constructor when `TCtx` is initialized. Implementations of this interface are provided by the user.
MemAllocator Interface - This is a simple root interface to make the TCtx interface reasonably generic so that different allocator approaches can be used in the future. It is passed to the TCtx constructor when TCtx is initialized. It is a low level allocator that does not know the type of an object being allocated. The allocators with this interface can also be used directly. In this case the user is responsible for the explicit deallocation of objects (with dealloc).

If the MemAllocator interface is passed as a parameter to the TCtx constructor, then, in many cases, it makes sense to overwrite the operator new. In this case all memory allocations in both C and C++ can be done by the same allocator.

Tctx Interface - This is an implicit interface to XML context implementations. It is primarily used for memory allocation, error (not exception) handling, and different encodings handling. The context interface is an implicit interface that is supposed to be used as type parameter. The name TCtx will be used as a corresponding type parameter name. Its actual substitutions are instantiations of implementations parameterized (templatized) by real context implementations. In the case of errors XmlException might be thrown.

All constructors create and initialize context implementations. In a multithreaded environment a separate context implementation has to be initialized for each thread.

IO Namespace

The IO namespace specifies interfaces for the different input and output options for all XML tools.

IO Datatypes

Datatype InputSourceType specifies different kinds of input sources supported currently. They include:

- ISRC_URI - Input is to be read from the specified URI.
- ISRC_FILE - Input is to be read from a file.
- ISRC_BUFFER - Input is to be read from a buffer.
- ISRC_DOM - Input is a DOM tree.
- ISRC_CSTREAM - Input is a C level stream.

IO Interfaces

URISource - This is an interface to inputs from specified URIs.
FileSource - This is an interface to inputs from a file.
BufferSource - This is an interface to inputs from a buffer.
DOMSource - This is an interface to inputs from a DOM tree.
CStreamSource - This is an interface to inputs from a C level stream.

Tools Package

Tools is the package (sub-space of OracleXml) for types and interfaces related to the creation and instantiation of Oracle XML tools.
Tools Interfaces

FactoryException - Specifies tool's factory exceptions. It is derived from XMLExceptions.

Factory - XML tools factory. Hides implementations of all XML tools and provides methods to create objects representing these tools based on their ID values.

Error Message Files

Error message files are provided in the mesg subdirectory. The messages files also exist in the $ORACLE_HOME/xdk/mesg directory. You can set the environment variable ORA_XML_MESG to point to the absolute path of the mesg subdirectory, although this not required.

See Also: Oracle Database XML C++ API Reference package Ctx APIs for C++
This chapter contains these topics:

- Introduction to Parser for C++
- DOM Namespace
- DOM Interfaces
- Parser Namespace
- Thread Safety
- XML Parser for C++ Usage
- XML Parser for C++ Default Behavior
- C++ Sample Files

Note: Use the new unified C++ API in xml.hpp for new XDK applications. The old C++ API in oraxml.hpp is deprecated and supported only for backward compatibility.

Introduction to Parser for C++

Oracle XML parser for C++ determines whether an XML document is well-formed and optionally validates it against a DTD or XML schema. The parser constructs an object tree which can be accessed through one of the following two XML APIs:

- DOM: Tree-based APIs. A tree-based API compiles an XML document into an internal tree structure, then allows an application to navigate that tree using the Document Object Model (DOM), a standard tree-based API for XML and HTML documents.

- SAX: Event-based APIs. An event-based API, on the other hand, reports parsing events (such as the start and end of elements) directly to the application through a user defined SAX even handler, and does not usually build an internal tree. The application implements handlers to deal with the different events, much like handling events in a graphical user interface.

Tree-based APIs are useful for a wide range of applications, but they often put a great strain on system resources, especially if the document is large (under very controlled circumstances, it is possible to construct the tree in a lazy fashion to avoid some of this problem). Furthermore, some applications need to build their own, different data trees, and it is very inefficient to build a tree of parse nodes, only to map it onto a new tree.
DOM Namespace

This is the namespace for DOM-related types and interfaces.

DOM interfaces are represented as generic references to different implementations of the DOM specification. They are parameterized by `Node` that supports various specializations and instantiations. Of them, the most important is `xmlnode` which corresponds to the current C implementation.

These generic references do not have a NULL-like value. Any implementation must never create a reference with no state (like NULL). If there is a need to signal that something has no state, an exception should be thrown.

Many methods might throw the `SYNTAX_ERR` exception, if the DOM tree is incorrectly formed, or throw `UNDEFINED_ERR`, in the case of wrong parameters or unexpected NULL pointers. If these are the only errors that a particular method might throw, it is not reflected in the method signature.

Actual DOM trees do not depend on the context, `TCtx`. However, manipulations on DOM trees in the current, `xmlctx`-based implementation require access to the current context, `TCtx`. This is accomplished by passing the context pointer to the constructor of `DOMImplRef`. In multithreaded environment `DOMImplRef` is always created in the thread context and, so, has the pointer to the right context.

`DOMImplRef` provides a way to create DOM trees. `DomImplRef` is a reference to the actual `DOMImplementation` object that is created when a regular, non-copy constructor of `DomImplRef` is invoked. This works well in a multithreaded environment where DOM trees need to be shared, and each thread has a separate `TCtx` associated with it. This works equally well in a single threaded environment.

`DOMString` is only one of the encodings supported by Oracle implementations. The support of other encodings is an Oracle extension. The `oratext*` data type is used for all encodings.

Interfaces represent DOM level 2 Core interfaces according to [http://www.w3.org/TR/DOM-Level-2-Core/core.html](http://www.w3.org/TR/DOM-Level-2-Core/core.html). These C++ interfaces support the DOM specification as closely as possible. However, Oracle cannot guarantee that the specification is fully supported by our implementation because the W3C specification does not cover C++ binding.

DOM Datatypes

*DATATYPE DomNodeType* - Defines types of DOM nodes.

*DATATYPE DomExceptionCode* - Defines exception codes returned by the DOM API.

DOM Interfaces

*DOMException Interface* - See exception `DOMException` in the W3C DOM documentation. DOM operations only raise exceptions in "exceptional" circumstances: when an operation is impossible to perform (either for logical reasons, because data is lost, or because the implementation has become unstable). The functionality of `XMLException` can be used for a wider range of exceptions.

*NodeRef Interface* - See interface `Node` in the W3C documentation.

*DocumentRef Interface* - See interface `Document` in the W3C documentation.

*DocumentFragmentRef Interface* - See interface `DocumentFragment` in the W3C documentation.

*ElementRef Interface* - See interface `Element` in the W3C documentation.
DOM Namespace

AttrRef Interface - See interface Attr in the W3C documentation.
CharacterDataRef Interface - See interface CharacterData in the W3C documentation.
TextRef Interface - See Text nodes in the W3C documentation.
CDATASectionRef Interface - See CDATASection nodes in the W3C documentation.
CommentRef Interface - See Comment nodes in the W3C documentation.
ProcessingInstructionRef Interface - See PI nodes in the W3C documentation.
EntityRef Interface - See Entity nodes in the W3C documentation.
EntityReferenceRef Interface - See EntityReference nodes in the W3C documentation.
NotationRef Interface - See Notation nodes in the W3C documentation.
DocumentTypeRef Interface - See DTD nodes in the W3C documentation.
DOMImplRef Interface - See interface DOMImplementation in the W3C DOM documentation. DOMImplementation is fundamental for manipulating DOM trees. Every DOM tree is attached to a particular DOM implementation object. Several DOM trees can be attached to the same DOM implementation object. Each DOM tree can be deleted and deallocated by deleting the document object. All DOM trees attached to a particular DOM implementation object are deleted when this object is deleted. DOMImplementation object is not visible to the user directly. It is visible through class DOMImplRef. This is needed because of requirements in the case of multithreaded environments

NodeListRef Interface - Abstract implementation of node list. See interface NodeList in the W3C documentation.
NamedNodeMapRef Interface - Abstract implementation of a node map. See interface NamedNodeMap in the W3C documentation.

DOM Traversal and Range Datatypes

DATATYPE AcceptNodeCode defines values returned by node filters provided by the user and passed to iterators and tree walkers.
DATATYPE WhatToShowCode specifies codes to filter certain types of nodes.
DATATYPE RangeExceptionCode specifies Exception kinds that can be thrown by the Range interface.
DATATYPE CompareHowCode specifies kinds of comparisons that can be done on two ranges.

DOM Traversal and Range Interfaces

NodeFilter Interface - DOM 2 Node Filter.
NodeIterator Interface - DOM 2 Node Iterator.
TreeWalker Interface - DOM 2 TreeWalker.
DocumentTraversal Interface - DOM 2 interface.
RangeException Interface - Exceptions for DOM 2 Range operations.
Range Interface - DOM 2 Range.
DocumentRange Interface - DOM 2 interface.
Parser Namespace

DOMParser Interface - DOM parser root class.
GParser Interface - Root class for XML parsers.
ParserException Interface - Exception class for parser and validator.
SAXHandler Interface - Root class for current SAX handler implementations.
SAXHandlerRoot Interface - Root class for all SAX handlers.
SAXParser Interface - Root class for all SAX parsers.
SchemaValidator Interface - XML schema-aware validator.

GParser Interface

GParser Interface - Root class for all XML parser interfaces and implementations. It is not an abstract class, that is, it is not an interface. It is a real class that allows users to set and check parser parameters.

DOMParser Interface

DOMParser Interface - DOM parser root abstract class or interface. In addition to parsing and checking that a document is well formed, DOMParser provides means to validate the document against DTD or XML schema.

SAXParser Interface

SAXParser Interface - Root abstract class for all SAX parsers.

SAX Event Handlers
To use SAX, a SAX event handler class should be provided by the user and passed to the SAXParser in a call to parse() or set before such call.
SAXHandlerRoot Interface - root class for all SAX handlers.
SAXHandler Interface - root class for current SAX handler implementations.

Thread Safety
If threads are forked off somewhere in the midst of the init-parse-term sequence of calls, you will get unpredictable behavior and results.

XML Parser for C++ Usage

1. A call to Tools::Factory to create a parser initializes the parsing process.
2. The XML input can be any of the InputSource kinds (see IO namespace).
3. DOMParser invocation results in the DOM tree.
4. SAXParser invocation results in SAX events.
5. A call to parser destructor terminates the process.

XML Parser for C++ Default Behavior

The following is the XML parser for C++ default behavior:
- Character set encoding is UTF-8. If all your documents are ASCII, you are encouraged to set the encoding to US-ASCII for better performance.

- Messages are printed to stderr unless msghdlr is specified.

- XML parser for C++ determines whether an XML document is well-formed and optionally validates it against a DTD. The parser constructs an object tree that can be accessed through a DOM interface or operates serially through a SAX interface.

- A parse tree which can be accessed by DOM APIs is built unless saxcb is set to use the SAX callback APIs. Note that any of the SAX callback functions can be set to NULL if not needed.

- The default behavior for the parser is to check that the input is well-formed but not to check whether it is valid. The flag XML_FLAG_VALIDATE can be set to validate the input. The default behavior for whitespace processing is to be fully conformant to the XML 1.0 spec, that is, all whitespace is reported back to the application but it is indicated which whitespace is ignorable. However, some applications may prefer to set the XML_FLAG_DISCARD_WHITESPACE which will discard all whitespace between an end-element tag and the following start-element tag.

---

**Note:** It is recommended that you set the default encoding explicitly if using only single byte character sets (such as US-ASCII or any of the ISO-8859 character sets) for performance up to 25% faster than with multibyte character sets, such as UTF-8.

---

- In both of these cases, an event-based API provides a simpler, lower-level access to an XML document: you can parse documents much larger than your available system memory, and you can construct your own data structures using your callback event handlers.

### C++ Sample Files

`xdk/demo/cpp/parser/` directory contains several XML applications to illustrate how to use the XML parser for C++ with the DOM and SAX interfaces.

Change directories to the sample directory (`$ORACLE_HOME/xdk/demo/cpp` on Solaris, for example) and read the README file. This will explain how to build the sample programs.

Table 25–1 lists the sample files in the directory. Each file `*Main.cpp` has a corresponding `*Gen.cpp` and `*Gen.hpp`.

<table>
<thead>
<tr>
<th>Sample File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMSampleMain.cpp</td>
<td>Sample usage of C++ interfaces of XML parser and DOM.</td>
</tr>
<tr>
<td>FullDOMSampleMain.cpp</td>
<td>Manually build DOM and then exercise.</td>
</tr>
<tr>
<td>SAXSampleMain.cpp</td>
<td>Source for SAXSample program.</td>
</tr>
</tbody>
</table>

**See Also:** *Oracle Database XML C++ API Reference* for parser package APIs for C++
This chapter contains these topics:

- Accessing XSLT for C++
- Xsl Namespace
- XSLT for C++ DOM Interface Usage
- Invoking XSLT for C++
- Using the Sample Files Included with the Software

**Note:** Use the new unified C++ API in `xml.hpp` for new XDK applications. The old C++ API in `oraxml.hpp` is deprecated and supported only for backward compatibility, but will not be enhanced. It will be removed in a future release.

### Accessing XSLT for C++

XSLT for C++ is provided with Oracle Database. Sample files are located at `xdk/demo/cpp/new/readme.html` in the root directory of the software archive contains release specific information including bug fixes and API additions.

**See Also:** "XVM Processor" on page 17-1

### Xsl Namespace

This is the namespace for XSLT compilers and transformers.

### Xsl Interfaces

- `XslException` Interface - Root interface for all XSLT-related exceptions.
- `Transformer` Interface - Basic XSLT processor. This interface can be used to invoke all XSLT processors.
- `CompTransformer` Interface - Extended XSLT processor. This interface can be used only with processors that create intermediate binary bytecode (currently XVM-based processor only).
- `Compiler` Interface - XSLT compiler. It is used for compilers that compile XSLT into binary bytecode.
XSLT for C++ DOM Interface Usage

1. There are two inputs to XMLParser.xmlparse():
   - The XML document
   - The stylesheet to be applied to the XML document
2. Any XSLT processor is initiated by calling the tools factory to create a particular XSLT transformer or compiler.
3. The stylesheet is supplied to any transformer by calling setXSL() member functions.
4. The XML instance document is supplied as a parameter to the transform member functions.
5. The resultant document (XML, HTML, VML, and so on) is typically sent to an application for further processing. The document is sent as a DOM tree or as a sequence of SAX events. SAX events are produced if a SAX event handler is provided by the user.
6. The application terminates the XSLT processors by invoking their destructors.

Invoking XSLT for C++

XSLT for C++ can be invoked in two ways:
- By invoking the executable on the command line
- By writing C++ code and using the supplied APIs

Command Line Usage

XSLT for C++ can be called as an executable by invoking bin/xml.

See Also: Table 18–4, "C XML Parser Command-Line Options"

Writing C++ Code to Use Supplied APIs

XSLT for C++ can also be invoked by writing code to use the supplied APIs. The code must be compiled using the headers in the public subdirectory and linked against the libraries in the lib subdirectory. Please see the Makefile or make.bat in xdk/demo/cpp/new for full details of how to build your program.

Using the Sample Files Included with the Software

The $ORACLE_HOME/xdk/demo/cpp/parser/ directory contains several XML applications to illustrate how to use the XSLT for C++.

Table 26–1 lists the sample files.
### Table 26–1  XSLT for C++ Sample Files

<table>
<thead>
<tr>
<th>Sample File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLSampleMain.cpp</td>
<td>Sources for sample XSLT usage program. XSLSample takes two arguments, the XSLT stylesheet and the XML file. If you redirect stdout of this program to a file, you may have some output missing, depending on your environment.</td>
</tr>
<tr>
<td>XSLSampleGen.cpp</td>
<td></td>
</tr>
<tr>
<td>XSLSampleGen.hpp</td>
<td></td>
</tr>
<tr>
<td>XVMSampleMain.cpp</td>
<td>Sources for the sample XVM usage program.</td>
</tr>
<tr>
<td>XVMSampleGen.cpp</td>
<td></td>
</tr>
<tr>
<td>XVMSampleGen.hpp</td>
<td></td>
</tr>
</tbody>
</table>
This chapter contains these topics:

- Oracle XML Schema Processor for C++
- XML Schema Processor API
- Running the Provided XML Schema for C++ Sample Programs

---

**Oracle XML Schema Processor for C++**

The XML Schema processor for C++ is a companion component to the XML parser for C++ that allows support to simple and complex datatypes into XML applications.

The XML Schema processor for C++ supports the W3C XML Schema Recommendation. This makes writing custom applications that process XML documents straightforward, and means that a standards-compliant XML Schema processor is part of the XDK on each operating system where Oracle is ported.

**Oracle XML Schema for C++ Features**

XML Schema processor for C++ has the following features:

- Supports simple and complex types
- Built upon the XML parser for C++
- Supports the W3C XML Schema Recommendation

The XML Schema processor for C++ class is OracleXml::Parser::SchemaValidator.

**See Also:** Oracle Database XML C++ API Reference schema validator interface

---

**Online Documentation**

Documentation for Oracle XML Schema processor for C++ is located in /xdk/doc/cpp/schema directory in your install area.
Standards Conformance

The XML Schema processor for C++ conforms to the following standards:

- W3C recommendation for Extensible Markup Language (XML) 1.0
- W3C recommendation for Document Object Model Level 1.0
- W3C recommendation for Namespaces in XML 1.0
- W3C recommendation for XML Schema 1.0

XML Schema Processor API

Interface SchemaValidator is an abstract template class to handle XML schema-based validation of XML documents.

Invoking XML Schema Processor for C++

The XML Schema processor for C++ can be called as an executable by invoking bin/schema in the install area. This takes the arguments:

- XML instance document
- Optionally, a default schema
- Optionally, the working directory

Table 27–1 lists the options (can be listed if the option is invalid or -h is the option):

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0</td>
<td>Always exit with code 0 (success).</td>
</tr>
<tr>
<td>-e encoding</td>
<td>Specify default input file encoding.</td>
</tr>
<tr>
<td>-E encoding</td>
<td>Specify output/data/presentation encoding.</td>
</tr>
<tr>
<td>-h</td>
<td>Help. Prints these choices.</td>
</tr>
<tr>
<td>-i</td>
<td>Ignore provided schema.</td>
</tr>
<tr>
<td>-o num</td>
<td>Validation option.</td>
</tr>
<tr>
<td>-p</td>
<td>Print document instance to stdout on success.</td>
</tr>
<tr>
<td>-u</td>
<td>Force the Unicode path.</td>
</tr>
<tr>
<td>-v</td>
<td>Version - display version, then exit.</td>
</tr>
</tbody>
</table>

Table 27–1 XML Schema Processor for C++ Command Line Options

The XML Schema processor for C++ can also be invoked by writing code using the supplied APIs. The code must be compiled using the headers in the include subdirectory and linked against the libraries in the lib subdirectory. See Makefile or Make.bat in the xdk/demo/cpp/schema directory for details on how to build your program.

Error message files in different languages are provided in the mesg subdirectory.

Running the Provided XML Schema for C++ Sample Programs

The directory xdk/demo/cpp/schema contains a sample application that illustrates how to use Oracle XML Schema processor for C++ with its API. Table 27–2 lists the sample files provided.
To build the sample programs, run `make`.

To build the programs and run them, comparing the actual output to expected output:

```
make sure
```
This chapter contains these topics:

- XPath Interfaces
- Sample Programs

---

**Note:** Use the new unified C++ API in `xml.hpp` for new XDK applications. The old C++ API in `oraxml.hpp` is deprecated and supported only for backward compatibility, but will not be enhanced. It will be removed in a future release.

---

**XPath Interfaces**

- **Processor Interface** - basic XPath processor interface that any XPath processor is supposed to conform to.
- **CompProcessor Interface** - extended XPath processor that adds an ability to use XPath expressions pre-compiled into an internal binary representation. In this release this interface represents Oracle virtual machine interface.
- **Compiler Interface** - XPath compiler into binary representation.
- **NodeSetRef Interface** - defines references to node sets when they are returned by the XPath expression evaluation.
- **XPathException Interface** - exceptions for XPath compilers and processors.
- **XPathObject Interface** - interface for XPath 1.0 objects.

**Sample Programs**

Sample programs are located in `xdk/demo/cpp/new`.

The programs `XslXPathSample` and `XvmXPathSample` have sources:

- `XslXPathSampleGen.hpp`, `XslXPathSampleGen.cpp`, `XslXPathSampleMain.cpp`, `XslXPathSampleForce.cpp`
- `XvmXPathSampleGen.hpp`, `XvmXPathSampleGen.cpp`, `XvmXPathSampleMain.cpp`, `XvmXPathSampleForce.cpp`

**See Also:** *Oracle Database XML C++ API Reference* package XPATH APIs for C++
This chapter contains these topics:

- Accessing XML C++ Class Generator
- Using XML C++ Class Generator
- Using the XML C++ Class Generator Command-Line Utility
- Using the XML C++ Class Generator Examples

**Accessing XML C++ Class Generator**

The XML C++ class generator is provided with Oracle Database.

**Using XML C++ Class Generator**

The XML C++ class generator creates source files from an XML DTD or XML Schema. The class generator takes the Document Type Definition (DTD) or the XML Schema, and generates classes for each defined element. Those classes are then used in a C++ program to construct XML documents conforming to the DTD.

This is useful when an application wants to send an XML message to another application based on an agreed-upon DTD or XML Schema, or as the back end of a Web form to construct an XML document. Using these classes, C++ applications can construct, validate, and print XML documents that comply with the input.

The class generator works in conjunction with the Oracle XML parser for C++, which parses the input and passes the parsed document to the class generator.

**External DTD Parsing**

The XML C++ class generator can also parse an external DTD directly without requiring a complete (dummy) document by using the Oracle XML parser for C++ routine `xmlparsedtd()`.

The provided command-line program `xmlcg` has a `-d` option that is used to parse external DTDs.

**Error Message Files**

Error message files are provided in the `mesg/` subdirectory. The messages files also exist in the `$ORACLE_HOME/xdk/mesg` directory. You may set the environment variable `ORA_XML_MESG` to point to the absolute path of the `mesg` subdirectory although this is not required.
Using the XML C++ Class Generator Command-Line Utility

The standalone class generator can be called as an executable by invoking `bin/xmlcg`.

1. You can run the C++ class generator from the command line as follows:

   ```bash
   xmlcg [options] input_file
   ```

   Table 29-1 describes the options for the utility.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d name</td>
<td>Input is an external DTD or a DTD file. Generates <code>name.cpp</code> and <code>name.h</code>.</td>
</tr>
<tr>
<td>-o directory</td>
<td>Output directory for generated files (the default is the current directory).</td>
</tr>
<tr>
<td>-e encoding</td>
<td>Default input file encoding.</td>
</tr>
<tr>
<td>-h</td>
<td>Show this usage help.</td>
</tr>
<tr>
<td>-v</td>
<td>Show the class generator version validator options.</td>
</tr>
<tr>
<td>-s name</td>
<td>Input is an XML Schema file with the given name. Generates <code>name.cpp</code> and <code>name.h</code>.</td>
</tr>
</tbody>
</table>

`input_file` name is the name of the parsed XML document with `<!DOCTYPE>` definitions, or parsed DTD, or an XML Schema document. The XML document must have an associated DTD.

The DTD input to the XML C++ class generator is an XML document containing a DTD, or an external DTD. The document body itself is ignored; only the DTD is relevant, though the document must conform to the DTD.

2. If invalid options, or no input is provided, a usage message with the preceding information is output.

3. Two source files are output, a `name.h` header file and a C++ file, `name.cpp`. These are named after the DTD file.

4. The output files are typically used to generate XML documents.

Constructors are provided for each class (element) that allow an object to be created in the following two ways:

- Initially empty, then adding the children or data after the initial creation
- Created with the initial full set of children or initial data

A method is provided for `#PCDATA` (and Mixed) elements to set the data and, when appropriate, set an element's attributes.

Input to the XML C++ Class Generator

The input is an XML document containing a DTD. The document body itself is ignored; only the DTD is relevant, though the dummy document must conform to the DTD. The underlying XML parser only accepts file names for the document and associated external entities.

Using the XML C++ Class Generator Examples

Table 29-2 lists the demo XML C++ class generator files:
The `make.bat` batch file (on Windows) or `Makefile` (on UNIX) do the following:

- Generate classes based on `CG.xml` into `Sample.h` and `Sample.cpp`
- Compile the program `CG.cpp` (using `Sample.h`), and link this with the `Sample` object into an executable named `CG.exe` in the `bin` (or `.../bin`) directory.

### XML C++ Class Generator Example 1: XML — Input File to Class Generator, CG.xml

This XML file, `CG.xml`, inputs XML C++ class generator. It references the DTD file, `CG.dtd`.

```xml
<?xml version="1.0"?>
<!DOCTYPE Sample SYSTEM "CG.dtd">
<Sample>
  <B>Be!</B>
  <D attr="value"></D>
  <E>
    <F>Formula1</F>
    <F>Formula2</F>
  </E>
</Sample>
```

### XML C++ Class Generator Example 2: DTD — Input File to Class Generator, CG.dtd

This DTD file, `CG.dtd` is referenced by the XML file `CG.xml`. `CG.xml` inputs XML C++ class generator.

```xml
<!ELEMENT Sample (A | (B, (C | (D, E))) | F)>
<!ELEMENT A (#PCDATA)>
<!ELEMENT B (#PCDATA | F)*)
<!ELEMENT C (#PCDATA)>
<!ELEMENT D (#PCDATA)>
<!ATTLIST D attr CDATA #REQUIRED>
<!ELEMENT E (F, F)>
<!ELEMENT F (#PCDATA)>
```

### XML C++ Class Generator Example 3: CG Sample Program

The CG sample program, `CG.cpp`, does the following:

1. Initializes the XML parser.
2. Loads the DTD (by parsing the DTD-containing file— the dummy document part is ignored).
3. Creates some objects using the generated classes.

---

**Table 29–2 XML C++ Class Generator Files**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.cpp</td>
<td>Sample program</td>
</tr>
<tr>
<td>CG.xml</td>
<td>XML file contains DTD and dummy document</td>
</tr>
<tr>
<td>CG.dtd</td>
<td>DTD file referenced by CG.xml</td>
</tr>
<tr>
<td>Make.bat on Windows</td>
<td>Batch file (on Windows) or Make file (on UNIX) to generate classes and build the sample programs.</td>
</tr>
<tr>
<td>Makefile on UNIX</td>
<td></td>
</tr>
<tr>
<td>README</td>
<td>A readme file with these instructions</td>
</tr>
</tbody>
</table>
4. Invokes the validation function which verifies that the constructed classes match the DTD.

5. Writes the constructed document to Sample.xml.

```
#ifndef ORAXMLDOM_ORACLE
#include <oraxmldom.h>
#endif

#include <fstream.h>
#include "Sample.h"
#define DTD_DOCUMENT "CG.xml"
#define OUT_DOCUMENT Sample.xml"

int main()
{
    XMLParser parser;
    Document *doc;
    Sample  *samp;
    B        *b;
    D        *d;
    E        *e;
    F        *f1, *f2;
    fstream  *out;
    ub4      flags = XML_FLAG_VALIDATE;
    uword    ecode;

    // Initialize XML parser
    cout << "Initializing XML parser...\n";
    if (ecode = parser.xmlinit())
    {
        cout << "Failed to initialize parser, code " << ecode << "\n";
        return 1;
    }

    // Parse the document containing a DTD; parsing just a DTD is not possible yet, so the file must contain a valid document (which is parsed but we're ignoring).
    cout << "Loading DTD from " << DTD_DOCUMENT << "...\n";
    if (ecode = parser.xmlparse((oratext *) DTD_DOCUMENT, (oratext *)0, flags))
    {
        cout << "Failed to parse DTD document " << DTD_DOCUMENT << ", code " << ecode << "\n";
        return 2;
    }

    // Fetch dummy document
    cout << "Fetching dummy document...\n";
    doc = parser.getDocument();

    // Create the constituent parts of a Sample
    cout << "Creating components...\n";
```
b = new B(doc, (String) "Be there or be square");
d = new D(doc, (String) "Dit dah");
d->setattr((String) "attribute value");
f1 = new F(doc, (String) "Formula1");
f2 = new F(doc, (String) "Formula2");
e = new E(doc, f1, f2);

// Create the Sample
cout << "Creating top-level element...\n";
samp = new Sample(doc, b, d, e);

// Validate the construct
cout << "Validating...\n";
if (ecode = parser.validate(samp))
{
    cout << "Validation failed, code " << ecode << "\n";
    return 3;
}

// Write out doc
cout << "Writing document to " << OUT_DOCUMENT << "\n";
if (!(out = new ofstream(OUT_DOCUMENT, ios::out)))
{
    cout << "Failed to open output stream\n";
    return 4;
}
samp->print(out, 0);
out->close();

// Everything's OK
cout << "Success.\n";

// Shut down
parser.xmlterm();
return 0;

// end of CG.cpp
This part contains the following reference chapters for the XDK:

- Chapter 30, "XSQL Pages Reference"
- Chapter 31, "XDK Standards"
- Appendix A, "Oracle XDK for Java Error Messages"
This chapter contains reference information for the XSQL pages framework. "XSQL Configuration File Parameters" on page 30-2 describes settings in the XSQL configuration file. Table 30–1 lists the legal built-in actions for XSQL pages.

<table>
<thead>
<tr>
<th>XSQL Action Element</th>
<th>Handler Class in oracle.xml.xsql.actions</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="">xsql:action</a></td>
<td>XSQLExtensionActionHandler</td>
<td>Invoke a user-defined action handler, implemented in Java, for executing custom logic and including custom XML data in your XSQL page.</td>
</tr>
<tr>
<td><a href="">xsql:delete-request</a></td>
<td>XSQLDeleteRequestHandler</td>
<td>Delete an existing row in the database based on the posted XML document supplied in the request.</td>
</tr>
<tr>
<td><a href="">xsql:dml</a></td>
<td>XSQLDMLHandler</td>
<td>Execute a SQL DML statement or a PL/SQL anonymous block.</td>
</tr>
<tr>
<td><a href="">xsql:if-param</a></td>
<td>XSQLIfParamHandler</td>
<td>Conditionally include XML content or other XSQL actions.</td>
</tr>
<tr>
<td><a href="">xsql:include-owa</a></td>
<td>XSQLIncludeOWAHandler</td>
<td>Include the results of a stored procedure that uses the Oracle Web Agent (OWA) packages in the database to generate XML.</td>
</tr>
<tr>
<td><a href="">xsql:include-param</a></td>
<td>XSQLGetParameterHandler</td>
<td>Include a parameter and its value as an element in the XSQL page.</td>
</tr>
<tr>
<td><a href="">xsql:include-posted-xml</a></td>
<td>XSQLIncludePostedXMLHandler</td>
<td>Include the XML document that has been posted in the request into the XSQL page.</td>
</tr>
<tr>
<td><a href="">xsql:include-request-params</a></td>
<td>XSQLIncludeRequestHandler</td>
<td>Include all request parameters as XML elements in the XSQL page.</td>
</tr>
<tr>
<td><a href="">xsql:include-xml</a></td>
<td>XSQLIncludeXMLHandler</td>
<td>Include arbitrary XML resources at any point in your page by relative or absolute URL.</td>
</tr>
<tr>
<td><a href="">xsql:include-xsql</a></td>
<td>XSQLIncludeXSQLHandler</td>
<td>Include the results of one XSQL page at any point inside another.</td>
</tr>
<tr>
<td><a href="">xsql:insert-param</a></td>
<td>XSQLInsertParameterHandler</td>
<td>Insert the XML document contained in the value of a single parameter.</td>
</tr>
<tr>
<td><a href="">xsql:insert-request</a></td>
<td>XSQLInsertRequestHandler</td>
<td>Insert the XML document or HTML form posted in the request into a table or view.</td>
</tr>
</tbody>
</table>
You can use the XSQL configuration file to tune your XSQL pages environment. Table 30–2 defines the legal parameters.

**Table 30–2 XSQL Configuration File Settings**

<table>
<thead>
<tr>
<th>Configuration Setting Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSQLConfig/servlet/output-buffer-size</td>
<td>Sets the size in bytes of the buffered output stream. If the servlet engine already buffers I/O to the servlet output stream, you can set to 0 (the default) to avoid additional buffering. Any non-negative integer is valid.</td>
</tr>
<tr>
<td>XSQLConfig/servlet/suppress-mime-charset/media-type</td>
<td>The XSQL servlet sets the HTTP <strong>ContentType</strong> header to indicate the MIME type of the resource returned to the request. By default, the servlet includes the optional character set data in the MIME type. For a particular MIME type, you can suppress the inclusion of the character set data by including a <code>&lt;media-type&gt;</code> element, with the desired MIME type as its contents. You can list any number of <code>&lt;media-type&gt;</code> elements. Valid value is any string.</td>
</tr>
<tr>
<td>XSQLConfig/processor/character-set-conversion/default-charset</td>
<td>Performs character set conversion by default on the value of HTTP parameters to compensate for the default character set used by most servlet engines. The default base character set used for conversion is the Java <strong>8859_1</strong>, which corresponds to the IANA <strong>ISO-8859-1</strong> set. If your servlet engine uses a different character set as its base, then you can specify this value here. To suppress character set conversion, specify the empty element <code>&lt;none/&gt;</code> as the content of the <code>&lt;default-charset&gt;</code> element instead of a character set name. This technique is useful if you are working with parameter values that are correctly representable with your servlet default character set. It eliminates overhead associated with performing the character set conversion. Valid values are any Java character set name or <code>&lt;none/&gt;</code>.</td>
</tr>
</tbody>
</table>
Table 30–2 (Cont.) XSQL Configuration File Settings

<table>
<thead>
<tr>
<th>Configuration Setting Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSQLConfig/processor/reload-connections-on-error</td>
<td>Connection definitions are cached when the XSQL pages processor is initialized. Set to yes (default) to cause the processor to reread the XSQLConfig.xml file to reload connection definitions if an attempt is made to request a connection name that is not in the cached connection list. The yes setting is useful for adding new &lt;connection&gt; definitions to the file while the servlet is running. Set to no to avoid reloading the connection definition file when a connection name is not found in the in-memory cache. Valid values are yes and no.</td>
</tr>
<tr>
<td>XSQLConfig/processor/default-fetch-size</td>
<td>Sets the default value of the row fetch size for retrieving information from SQL queries. It only takes effect when you use the Oracle JDBC driver; otherwise the setting is ignored. This technique reduces network round trips to the database from the servlet engine running in a different tier. Default is 50. Valid value is any nonzero positive integer.</td>
</tr>
<tr>
<td>XSQLConfig/processor/page-cache-size</td>
<td>Sets the size of the cache for XSQL page templates and so determines the maximum number of XSQL pages that are cached. Least recently used pages move out of the cache if you go above this number. Default is 25. Any nonzero positive integer is valid.</td>
</tr>
<tr>
<td>XSQLConfig/processor/stylesheet-cache-size</td>
<td>Sets the size of the cache for XSLT stylesheets and so determines the maximum number of XSQL pages that are cached. Least recently used pages move out of the cache if you go above this number. Default is 25. Any nonzero positive integer is valid.</td>
</tr>
<tr>
<td>XSQLConfig/processor/stylesheet-pool/initial</td>
<td>Each cached stylesheet is a pool of cached stylesheet instances to improve throughput. Sets the initial number of stylesheets to be allocated in each stylesheet pool. Default is 1. Valid value is any nonzero positive integer.</td>
</tr>
<tr>
<td>XSQLConfig/processor/stylesheet-pool/increment</td>
<td>Sets the number of stylesheets allocated when the stylesheet pool must grow due to increased load on the server. Default is 1. Valid value is any nonzero positive integer.</td>
</tr>
<tr>
<td>XSQLConfig/processor/stylesheet-pool/timeout-seconds</td>
<td>Sets the number of seconds of inactivity before a stylesheet instance in the pool is removed to free resources as the pool tries to shrink back to its initial size. Default is 60. Valid value is any nonzero positive integer.</td>
</tr>
<tr>
<td>XSQLConfig/processor/connection-pool/initial</td>
<td>Controls the initial number of JDBC connections allocated in each connection pool. The XSQL pages processor’s default connection manager implements connection pooling to improve throughput. Default is 2. Valid value is any nonzero positive integer.</td>
</tr>
<tr>
<td>XSQLConfig/processor/connection-pool/increment</td>
<td>Sets the number of connections allocated when the connection pool must grow due to increased load on the server. Default is 1. Valid value is any nonzero positive integer.</td>
</tr>
<tr>
<td>XSQLConfig/processor/connection-pool/timeout-seconds</td>
<td>Sets the number of seconds of inactivity before a JDBC connection in the pool is removed to free resources as the pool tries to shrink back to its initial size. Default is 60. Valid value is any nonzero positive integer.</td>
</tr>
<tr>
<td>XSQLConfig/processor/connection-pool/dump-allowed</td>
<td>Determines whether a diagnostic report of connection pool activity can be requested by passing the dump-pool=y parameter in the page request. Default is no. Valid value is yes or no.</td>
</tr>
<tr>
<td>XSQLConfig/processor/connection-manager/factory</td>
<td>Specifies the fully-qualified Java class name of the XSQL connection manager factory implementation. If not specified, default is XSQLConnectionManagerFactoryImpl. Valid value is any class name that implements the XSQLConnectionManagerFactory interface.</td>
</tr>
</tbody>
</table>
### Table 30-2 (Cont.) XSQL Configuration File Settings

<table>
<thead>
<tr>
<th>Configuration Setting Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>XSQLConfig/processor/owa/fetch-style</strong></td>
<td>Sets the default OWA Page Buffer fetch style used by the <code>&lt;xsql:include-owa&gt;</code> action. Valid values are CLOB (default) or TABLE. If set to CLOB, then the processor uses a temporary CLOB to retrieve the OWA page buffer. If set to TABLE, then the processor uses a more efficient approach that requires the Oracle database user-defined type named <code>XSQL_OWA_ARRAY</code>. Create this type with the following DDL statement: <code>CREATE TYPE xsql_owa_array AS TABLE OF VARCHAR2(32767)</code></td>
</tr>
<tr>
<td><strong>XSQLConfig/processor/timing/page</strong></td>
<td>Determines whether the XSQL page processor adds an <code>xsql-timing</code> attribute to the document element of the page whose value reports the elapsed number of milliseconds required to process the page. Valid values are yes or no (default).</td>
</tr>
<tr>
<td><strong>XSQLConfig/processor/timing/action</strong></td>
<td>Determines whether the XSQL page processor adds comment to the page just before the action element whose contents reports the elapsed number of milliseconds required to process the action. Valid values are yes or no (default).</td>
</tr>
<tr>
<td><strong>XSQLConfig/processor/logger/factory</strong></td>
<td>Specifies the fully-qualified Java class name of a custom XSQL logger factory implementation. If not set, then no logger is used. Valid value is any class name that implements the XSQLLoggerFactory interface.</td>
</tr>
<tr>
<td><strong>XSQLConfig/processor/error-handler/class</strong></td>
<td>Specifies the fully-qualified Java class name of a custom XSQL error handler. The specified handler is the default error handler implementation. If not set, then the default error handler is used. Valid value is any class name that implements the XSQLErrorHandler interface.</td>
</tr>
<tr>
<td><strong>XSQLConfig/processor/xml-parsing/preserve-whitespace</strong></td>
<td>Determines whether the XSQL pages processor preserves whitespace when parsing XSQL pages and XSLT stylesheets. Valid values are true (default) or false. Changing the default to false can slightly speed up processing of XSQL pages and stylesheets because ignoring whitespace while parsing is faster than preserving it.</td>
</tr>
<tr>
<td><strong>XSQLConfig/processor/security/stylesheet.defaults/allow-client-style</strong></td>
<td>Prevents client overriding of the stylesheet. Valid values are yes and no. During development it is sometimes useful to use the XSQL stylesheet override feature by providing a value for the <code>xml-stylesheet</code> parameter in the request. You can use the <code>xml-stylesheet=none</code> combination to temporarily disable the application of the stylesheet for debugging purposes. You can add the <code>allow-client-style=&quot;no&quot;</code> attribute to the document element of each XSQL page to prohibit client overriding of the stylesheet in production applications. This setting can globally change the default behavior for <code>allow-client-style</code> in a single place. This setting only specifies default behavior. If the attribute value is explicitly specified on the document element for a given XSQL page, its value takes precedence over this global default.</td>
</tr>
<tr>
<td><strong>XSQLConfig/processor/security/stylesheet.trusted-hosts/host</strong></td>
<td>Specifies any absolute URL to an XSLT stylesheet must be from a trusted host whose name is listed in the configuration file. List any number of <code>&lt;host&gt;</code> elements inside the <code>&lt;trusted-hosts&gt;</code> element. The name of the local machine, localhost, and <code>127.0.0.1</code> are trusted hosts by default. Valid values are any hostname or IP address. The XSLT processor supports Java extension functions. Typically, XSQL pages refer to XSLT stylesheets with relative URLs.</td>
</tr>
<tr>
<td><strong>XSQLConfig/http/proxyhost</strong></td>
<td>Sets the name of the HTTP proxy server to use when processing URLs with the HTTP protocol. Valid value is any hostname or IP address.</td>
</tr>
</tbody>
</table>
### Table 30–2 (Cont.) XSQL Configuration File Settings

<table>
<thead>
<tr>
<th>Configuration Setting Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSQLConfig/http/proxyport</td>
<td>Sets the port number of the HTTP proxy server to use when processing URLs with the HTTP protocol. Valid value is any nonzero integer.</td>
</tr>
<tr>
<td>XSQLConfig/connectiondefs/connection</td>
<td>Defines a short name and the JDBC details for a named connection used by the XSQL pages processor. You may supply any number of <code>&lt;connection&gt;</code> element children of <code>&lt;connectiondefs&gt;</code>. Each connection definition must supply a name attribute and may supply children elements <code>&lt;username&gt;</code>, <code>&lt;password&gt;</code>, <code>&lt;driver&gt;</code>, <code>&lt;dburl&gt;</code>, and <code>&lt;autocommit&gt;</code>.</td>
</tr>
<tr>
<td>XSQLConfig/connectiondefs/connection/username</td>
<td>Defines the username for the current connection.</td>
</tr>
<tr>
<td>XSQLConfig/connectiondefs/connection/password</td>
<td>Defines the password for the current connection.</td>
</tr>
<tr>
<td>XSQLConfig/connectiondefs/connection/dburl</td>
<td>Defines the JDBC connection URL for the current connection.</td>
</tr>
<tr>
<td>XSQLConfig/connectiondefs/connection/driver</td>
<td>Specifies the fully-qualified Java class name of the JDBC driver used for the current connection. If not specified, defaults to <code>oracle.jdbc.driver.OracleDriver</code>.</td>
</tr>
<tr>
<td>XSQLConfig/connectiondefs/connection/autocommit</td>
<td>Explicitly sets the Auto Commit flag for the current connection. If not specified, the connection uses the JDBC driver default setting for Auto Commit.</td>
</tr>
<tr>
<td>XSQLConfig/serializerdefs/serializer</td>
<td>Defines a named custom serializer implementation. You can supply any number of <code>&lt;serializer&gt;</code> element children of <code>&lt;serializerdefs&gt;</code>. Each must specify both a <code>&lt;name&gt;</code> and a <code>&lt;class&gt;</code> child element.</td>
</tr>
<tr>
<td>XSQLConfig/serializerdefs/serializer/name</td>
<td>Defines the name of the current custom serializer definition.</td>
</tr>
<tr>
<td>XSQLConfig/connectiondefs/connection/class</td>
<td>Specifies the fully-qualified Java class name of the current custom serializer. The class must implement the <code>XSQLDocumentSerializer</code> interface.</td>
</tr>
</tbody>
</table>
Purpose

Invokes a user-defined action handler, implemented in Java, for executing custom logic and including custom XML data in an XSQL page. The Java class invoked with this action must implement the oracle.xml.xsql.XSQLActionHandler interface.

Use <xsql:action> to perform tasks that are not handled by the built-in action handlers. Custom actions can supply arbitrary XML content to the data page and perform arbitrary processing.

Usage Notes

The XSQL page processor processes the actions in a page in the following way:

1. Constructs an instance of the action handler class with the default constructor.
2. Initializes the handler instance with the action element object and the page processor context by invoking the method init(Element actionElt, XSQLPageRequest context).
3. Invokes the method that allows the handler to handle the action handleAction(Node result).

Syntax

The syntax for this action is as follows, where handler is a single, required attribute named whose value is the fully-qualified Java class name of the invoked action, yourpackage is the Java package, and YourCustomHandler is the Java class:

```xml
<xsql:action handler="yourpackage.YourCustomHandler"/>
```

Some action handlers expect text content or element content to appear inside the <xsql:action> element. In this case, use syntax such as the following:

```xml
<xsql:action handler="yourpackage.YourCustomHandler">
  Some_text
</xsql:action>
```

You can also use the following syntax:

```xml
<xsql:action handler="yourpackage.YourCustomHandler">
  <some>
    <other/>
    <elements/>
    <here/>
  </some>
</xsql:action>
```

Attributes

The only required attribute is handler, but you can supply additional attributes to the handler. For example, if yourpackage.YourCustomHandler is expecting attributes named param1 and param2, then use the following syntax:

```xml
<xsql:action handler="yourpackage.YourCustomHandler" param1="xxx" param2="yyy"/>
```
Examples

Example 30–1 shows an XSQL page that invokes the myactions.StockQuotes Java class. It includes stock quotes from Google for any symbols passed in with the symbol parameter. If this parameter is not supplied, it supplies a default list.

Example 30–1  Retrieving Stock Quotes

```xml
<?xml version='1.0'?>
<page xmlns:urn:oracle-xsql>
  <xsql:action handler="myactions.StockQuotes"
    symbols="{@symbol}" symbol="ORCL,SAP,MSFT,IBM"/>
</page>
```
<xsql:delete-request>

Purpose

Accepts data posted from an XML document or HTML form and uses the XML SQL Utility (XSU) to delete the content of an XML document in canonical form from a target table or view.

By combining XSU with XSLT, you can transform XML into the canonical format expected by a given table. Afterward, you can use XSU to delete the resulting canonical XML. For a specified database table, the canonical XML form is given by one row of XML output from a SELECT * query against the table.

Syntax

The syntax for this action is as follows, where table_name is the name of a table and key is a list of one or more columns to use as the unique key:

<xsql:delete-request table='table_name' key-columns='key'/>

Attributes

Table 30–3 lists the optional attributes that you can use on the <xsql:delete-request> action. Required attributes are in bold.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table = &quot;string&quot;</td>
<td>Name of the table, view, or synonym to use for deleting the XML data.</td>
</tr>
<tr>
<td>key-columns = &quot;string string ...&quot;</td>
<td>Space-delimited or comma-delimited list of one or more column names. The processor uses the values of these names in the posted XML document to identify the existing rows to delete.</td>
</tr>
<tr>
<td>transform = &quot;URL&quot;</td>
<td>Relative or absolute URL of the XSLT transformation to use to transform the document to be deleted into canonical ROWSET/ROW format.</td>
</tr>
<tr>
<td>columns = &quot;string&quot;</td>
<td>Relative or absolute URL of the XSLT transformation to use to transform the document to be deleted into canonical ROWSET/ROW format.</td>
</tr>
<tr>
<td>commit = &quot;boolean&quot;</td>
<td>If set to yes (default), calls COMMIT on the current connection after a successful execution of the deletion. Valid values are yes and no.</td>
</tr>
<tr>
<td>commit-batch-size = &quot;integer&quot;</td>
<td>If a positive, nonzero integer is specified, then after each batch of integer deleted records, the processor issues a COMMIT. The default batch size is zero (0) if not specified, which means that the processor does not commit interim batches.</td>
</tr>
<tr>
<td>date-format = &quot;string&quot;</td>
<td>Date format mask to use for interpreting date field values in XML being deleted. Valid values are those documented for the java.text.SimpleDateFormat class.</td>
</tr>
<tr>
<td>error-param = &quot;string&quot;</td>
<td>Name of a page-private parameter that must be set to the string Error if a non-fatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
</tbody>
</table>

Examples

Example 30–2 specifies that the posted XML document should be transformed with the style.xsl stylesheet and then deleted from the departments table. The departments.department_id column is the primary key for the deletion.

Example 30–2  Deleting Rows

<?xml version="1.0"?>
<xsql:delete-request table="departments" transform="style.xsl"
connection="demo" key-columns="department_id" xmlns:xsql="urn:oracle-xsql"/>
<xsql:dml>

### Purpose

Executes a DML or DDL statement or a PL/SQL block. Typically, you use this tag to include statements that would be executed or rolled back together.

This action requires a database connection provided as a `connection="connnname"` attribute on the document element of the XSQL page in which it appears.

### Usage Notes

You cannot set parameter values by binding them in the position of OUT variables with `<xsql:dml>`. Only IN parameters are supported for binding.

### Syntax

The syntax for the action is as follows, where `DML_DDL_or_PLSQL` is a placeholder for a legal DML statement, DDL statement, or PL/SQL block:

```xml
<xsql:dml>
  DML_DDL_or_PLSQL
</xsql:dml>
```

### Attributes

Table 30–4 lists the optional attributes that you can use on the `<xsql:dml>` action.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>commit = &quot;boolean&quot;</code></td>
<td>If set to <code>yes</code>, calls commit on the current connection after a successful execution of the DML statement. Valid values are <code>yes</code> and <code>no</code> (default).</td>
</tr>
<tr>
<td><code>bind-params = &quot;string&quot;</code></td>
<td>Ordered, space-delimited list of one or more XSQL parameter names. The values of these parameters are used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.</td>
</tr>
<tr>
<td><code>error-param = &quot;string&quot;</code></td>
<td>Name of a page-private parameter that must be set to the string 'Error' if a nonfatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
<tr>
<td><code>error-statement = &quot;boolean&quot;</code></td>
<td>If set to <code>no</code>, suppresses the inclusion of the offending SQL statement in any <code>&lt;xsql-error&gt;</code> element generated. Valid values are <code>yes</code> (default) and <code>no</code>.</td>
</tr>
</tbody>
</table>

### Examples

**Example 30–3** inserts the username stored in the `webuser` cookie into a `request_log` table. Using bind variables guards against SQL injection attacks.

**Example 30–3 Inserting a Username into a Table**

```xml
<xsql:dml connection="demo" bind-params="webuser"
  xmlns:xsql="urn:oracle-xsql">
BEGIN
  INSERT INTO request_log(page,userid)
  VALUES( 'somepage.xsql', ? );
  COMMIT;
END;
</xsql:dml>
```
<xsql:if-param>

Purpose

Enables you to include elements and actions nested inside if a specified condition is true. If the condition is true, then all nested XML content and actions are included in the page. If the condition is false, then none of the nested XML content or actions is included (and thus none of the nested actions is executed).

Specify which parameter value is evaluated by supplying the required name attribute. Both simple parameter names as well as array-parameter names are supported.

Note: If the parameter being tested does not exist, the test evaluates to false.

Syntax

The syntax for the action is the following, where some_name is the value of the name attribute and test_condition is exactly one of the conditions listed in Table 30–5:

```xml
<xsql:if-param name="some_name" test_condition>
    element_or_action
</xsql:if-param>
```

Any XML content or XSQL action elements can be nested inside an <xsql:if-param>, including other <xsql:if-param> elements.

Attributes

In addition to the required name attribute, you must pick exactly one of the attributes listed in Table 30–5 to indicate how the parameter value (or values, in the array case) is tested. As with other XSQL actions, the attributes of the <xsql:if-param> action can contain lexical substitution parameter expressions such as {@paramName}.

Table 30–5  Attributes for <xsql:if-param>

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists=&quot;yes_or_no&quot;</td>
<td>If set to exists='yes', then this condition tests whether the named parameter exists and has a non-empty value. For an array-valued parameter, it tests whether the array-parameter exists and has at least one non-empty element. If set to exists='no', then this condition evaluates to true if the parameter does not exist, or if it exists but has an empty value. For an array-valued parameter, it evaluates to true if the parameter does not exist, or if all of the array elements are empty.</td>
</tr>
<tr>
<td>equals=&quot;stringValue&quot;</td>
<td>This condition tests whether the named parameter equals the string value provided. By default the comparison is an exact string match. For a case-insensitive match, supply the additional ignore-case=&quot;yes&quot; attribute as well. For an array-valued parameter, the condition tests whether any element in the array has the indicated value.</td>
</tr>
</tbody>
</table>
Examples

To test whether two different conditions are true, you can use nested `<xsql:if-param>` elements as shown in Example 30–4.

Example 30–4  Testing Conditions

```xml
<?xml version="1.0"?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<page connection='demo' xmlns:xsql='urn:oracle-xsql'>
<!--
| Set page parameter 'some_param' to value "some_value" if parameter 'a'
| exists, and if parameter 'b' has a value equal to 'X'
--><xsql:if-param name='a' exists='yes'>
  <xsql:if-param name='b' equals='X'>
    <xsql:set-page-param name='some_param' value='some_value'/>
  </xsql:if-param>
</xsql:if-param>
</xml:if-param>
</page>
```
<xsql:include-owa>

Purpose

Includes XML content generated by a database stored procedure. This action requires a database connection to be provided by supplying a `connection="connname"` attribute on the document element of the XSQL page in which it appears.

The stored procedure uses the standard Oracle Web Agent (OWA) packages (`HTP` and `HTF`) to "print" the XML tags into the server-side page buffer. Afterwards, the XSQL pages processor fetches, parses, and includes the dynamically-produced XML content in the data page. The stored procedure must generate a well-formed XML page or an appropriate error is displayed.

Usage Notes

You can create a wrapper procedure that constructs XML elements with the HTP package. Your XSQL page can invoke the wrapper procedure by using `<xsql:include-owa>`.

Syntax

The syntax for the action is as follows, where `PL/SQL_block` is a PL/SQL Block invoking a procedure that uses the HTP or HTF packages:

```xml
<xsql:include-owa>
  PL/SQL_block
</xsql:include-owa>
```

Attributes

Table 30–6 lists the optional attributes supported by this action.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bind-params = &quot;string&quot;</td>
<td>Ordered, space-delimited list of one or more XSQL parameter names. The values of these parameters are used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.</td>
</tr>
<tr>
<td>error-param = &quot;string&quot;</td>
<td>Name of a page-private parameter that must be set to the string 'Error' if a non-fatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
<tr>
<td>error-statement = 'boolean'</td>
<td>If set to no, suppresses the inclusion of the offending SQL statement in any <code>&lt;xsql-error&gt;</code> element generated. Valid values are yes (default) and no.</td>
</tr>
</tbody>
</table>

Examples

Assume that you write a PL/SQL procedure called `UpdateStatus` that updates the status of a project. The procedure uses HTP to print an `<UpdateStatus>` datagram that contains the element `<Success/>` if no errors occur or one or more `<Error>` elements if errors occur.

Example 30–5 shows how you can call `UpdateStatus` from an XSQL page. The example uses SQL bind variable instead of lexical substitution to prevent the possibility of SQL injection attacks.
**Example 30–5  Including XML Content Created by a Stored Procedure**

```xml
<sql:include-owa connection='demo'
     bind-params='project status'
     xmlns:xsql='urn:oracle-xsql'>
  UpdateStatus( ?,? );
</sql:include-owa>

Assume that a user enters an invalid status number for a project into a Web-based form. The form posts the input parameters to an XSQL page as shown in Example 30–5. The XSQL processor returns the following datagram, which an XSLT stylesheet could transform into an HTML error page:

```xml
<UpdateStatus>
  <Error Field="status">Status must be 1, 2, 3, or 4</Error>
</UpdateStatus>
```
Purpose

Includes an XML representation of the name and value of a single parameter. This technique is useful if an associated XSLT stylesheet needs to refer to parameter values with XPath expressions.

Syntax

The syntax of the action is as follows, where paramname is the name of a parameter:

```xml
<xsql:include-param name="paramname" />
```

The required name attribute supplies the name of the parameter whose value you want to include.

Attributes

The name attribute is required; there are no optional attributes.

Examples

Example 30–6 uses XPATH to obtain the value of a parameter and represent it in XML.

**Example 30–6 Including an XML Representation of a Parameter Value**

```xml
<?xml version='1.0'?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<page connection='demo' xmlns:xsql='urn:oracle-xsql' xmlns:p='http://www.companysite.com/products'>
  <xsql:set-page-param name='productid' xpath='/p:Products/productid'/>
  <xsql:include-param name='productid'/>
</page>
```

The XML fragment included in the datagram will be as follows:

```xml
<productid>12345</productid>
```

Suppose that you use an array parameter name to indicate that you want to treat the value as an array, as in the following example:

```xml
<xsql:include-param name="productid[]"/>
```

The XML fragment reflects all of the array values, as shown in the following example:

```xml
<productid>
  <value>12345</value>
  <value>33455</value>
  <value>88199</value>
</productid>
```

In this array-parameter name scenario, if productid happens to be a single-valued parameter, then the fragment looks identical to a one-element array, as illustrated in the following example:

```xml
<productid>
  <value>12345</value>
</productid>
```
<xsql:include-posted-xml>

Purpose
Includes the posted XML document in the XSQL page. If the user posts an HTML form instead of an XML document, then the XML included is similar to that included by the <xsql:include-request-params> action.

Syntax
The syntax of the action is as follows:
<xsql:include-posted-xml/>

Attributes
None.

Examples
Example 30–7 shows a sample XSQL page that includes a posted XML document.

Example 30–7  Including Posted XML
<?xml version="1.0"?>
<!DOCTYPE page SYSTEM "xsql.dtd">
<?xml-stylesheet type="text/xsql" href="somepage.xsql"?>
<page connection='demo'
     xmlns:xsql='urn:oracle-xsql'>
  <xsql:include-posted-xml/>
</page>
<xsql:include-request-params/>

Purpose

Includes an XML representation of all parameters in the request in the datagram. The action element is replaced in the page at page-request time with a tree of XML elements that represents the parameters available to the request.

This technique is useful if an associated XSLT stylesheet needs to refer to request parameter values with XPath expressions.

Usage Notes

When processing pages through the XSQL servlet, the XML included takes the form shown in Example 30–8.

**Example 30–8 Including Request Parameters**

```xml
<request>
  <parameters>
    <paramname>value1</paramname>
    <ParamName2>value2</ParamName2>
    ...
  </parameters>
  <session>
    <sessVarName>value1</sessVarName>
    ...
  </session>
  <cookies>
    <cookieName>value1</cookieName>
    ...
  </cookies>
</request>
```

When you use the XSQL command-line utility or the XSQLRequest class, the XML takes the form shown in Example 30–11.

**Example 30–9 Including Request Parameters**

```xml
<request>
  <parameters>
    <paramname>value1</paramname>
    <ParamName2>value2</ParamName2>
    ...
  </parameters>
</request>
```

The technique enables you to distinguish request parameters from session parameters or cookies because its value is a child element of `<parameters>`, `<session>`, or `<cookies>`.

Syntax

The syntax of the action is as follows:

```xml
<xsql:include-request-params/>
```
Attributes

None.

Examples

Example 30–10 shows a sample XSQL page that includes all request parameters in the data page.

Example 30–10 Including Request Parameters

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsql" href="cookie_condition.xsl"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
  <xsql:include-request-params/>
</page>

The cookie_condition.xsl stylesheet chooses an output format based on whether the siteuser cookie is present. Example 30–11 shows a fragment of the stylesheet.

Example 30–11 Testing for Conditions in a Stylesheet

<xsl:choose>
  <xsl:when test="/page/request/cookies/siteuser">
    ...
  </xsl:when>
  <xsl:otherwise>
    ...
  </xsl:otherwise>
</xsl:choose>
**Purpose**

Includes the XML contents of a local, remote, or database-driven XML resource in your datagram. You can specify the resource by URL or SQL statement. The server can deliver a resource that is a static XML file or dynamically created XML from a programmatic resource such as a servlet or CGI program.

**Syntax**

The syntax for this action is as follows, where *URL* is a relative URL or an absolute, HTTP-based URL to retrieve XML from another Web site:

```xml
<xsql:include-xml href="URL"/>
```

Alternatively, you can use the following syntax, where *SQL_statement* is a SQL SELECT statement selecting a single row containing a single CLOB or VARCHAR2 column value:

```xml
<xsql:include-xml>
  SQL_statement
</xsql:include-xml>
```

The `href` attribute and SQL statement are mutually exclusive. If you provide one, then the other is not allowed.

**Attributes**

Table 30–7 lists the attributes supported by this action. Required attributes are in bold.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>href=&quot;URL&quot;</code></td>
<td>The absolute, relative, or parameterized URL of the XML resource to be included. The resource can be a static file dynamic source.</td>
</tr>
<tr>
<td><code>bind-params = &quot;string&quot;</code></td>
<td>Ordered, space-delimited list of one or more XSQL parameter names. The values for these name will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.</td>
</tr>
<tr>
<td><code>error-param = &quot;string&quot;</code></td>
<td>Name of a page-private parameter that must be set to the string 'Error' if a non-fatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
</tbody>
</table>

**Examples**

Example 30–12 includes an XML document retrieved by a database query. The XML content is a CLOB-valued member field of a user-defined type. The XML included must come from a VARCHAR2 or CLOB column, not an XMLType.

**Example 30–12 Including an XML Document**

```xml
<?xml version="1.0"?>
<xsql:include-xml bind-params="id" connection="demo"
  xmlns:xsql="urn:oracle-xsql">
  SELECT x.document.contents doc FROM xmldoc x
  WHERE x.docid = ?
</xsql:include-xml>
```
Purpose

Includes the XML output of one XSQL page in another page. You can create a page that assembles the contents—optionally transformed—from other XSQL pages.

Usage Notes

If the aggregated page contains an `<?xml-stylesheet?>` processing instruction, then this stylesheet is applied before the result is aggregated. Thus, you can use `<xsql:include-xsql>` to chain XSLT stylesheets.

When one XSQL page aggregates another page by using `<xsql:include-xsql>`, all request-level parameters are visible to the nested page. For pages processed by the XSQL Servlet, the visible data includes session-level parameters and cookies. None of the aggregating page's page-private parameters are visible to the nested page.

Syntax

The syntax for this action is as follows, where `XSQL_page` is a relative or absolute URL of an XSQL page to be included:

```xml
<xsql:include-xsql href="XSQL_page"/>
```

Attributes

Table 30–8 lists the attributes supported by this action. Required attributes are in bold; all others are optional.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>href=&quot;string&quot;</code></td>
<td>Relative or absolute URL of XSQL page to be included.</td>
</tr>
<tr>
<td><code>error-param = &quot;string&quot;</code></td>
<td>Name of a page-private parameter that must be set to the string Error if a non-fatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
<tr>
<td><code>reparse = &quot;boolean&quot;</code></td>
<td>Indicates whether output of the included XSQL page must be reparsed before it is included. Valid values are no (default) and yes. This attribute is useful if the included XSQL page selects the text of an XML document fragment that the including page wants to treat as elements.</td>
</tr>
</tbody>
</table>

Examples

Example 30–13 displays an XSQL page that lists discussion forum categories.

```xml
Example 30–13 Categories.xsql

<?xml version="1.0"?>
<xsql:query connection="demo" xmlns:xsql="urn:oracle-xsql">
  SELECT name
  FROM categories
  ORDER BY name
</xsql:query>
```
Example 30–14 shows how you can include the results of the page in Example 30–13 into a page that lists the ten most recent topics in the current forum.

**Example 30–14** TopTenTopics.xsql

```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
<top-ten-topics connection="demo" xmlns:xsql="urn:oracle-xsql">
  <topics>
    <xsql:query max-rows="10">
      SELECT subject
      FROM topics
      ORDER BY last_modified DESC
    </xsql:query>
  </topics>
  <categories>
    <xsql:include-xsql href="Categories.xsql"/>
  </categories>
</top-ten-topics>
```

You can also use `<xsql:include-xsql>` to apply an XSLT stylesheet to an included page. Assume that you write the following XSLT stylesheets:

- cats-as-html.xsl, which renders the topics in HTML
- cats-as-wml.xsl, which renders the topics in WML

One approach for catering to two different types of devices is to create different XSQL pages for each device. **Example 30–15** shows an XSQL page that aggregates Categories.xsql and applies the cats-as-html.xsl stylesheet.

**Example 30–15** HTMLCategories.xsql

```xml
<?xml version="1.0"?>
<!-- HTMLCategories.xsql -->
<?xml-stylesheet type="text/xsl" href="cats-as-html.xsl"?>
<xsql:include-xsql href="Categories.xsql" xmlns:xsql="urn:oracle-xsql"/>
```

**Example 30–16** shows an XSQL page that aggregates Categories.xsql and applies the cats-as-html.xsl stylesheet for delivering to wireless devices.

**Example 30–16** WMLCategories.xsql

```xml
<?xml version="1.0"?>
<!-- WMLCategories.xsql -->
<?xml-stylesheet type="text/xsl" href="cats-as-wml.xsl"?>
<xsql:include-xsql href="Categories.xsql" xmlns:xsql="urn:oracle-xsql"/>
```
<xsql:insert-param>

**Purpose**

Inserts the value of a parameter into a table or view. Use this tag when the client is posting a well-formed XML document as text in an HTTP parameter or individual HTML form field.

By combining the XML SQL Utility (XSU) with XSLT, you can transform XML into the canonical format expected by a given table. Afterward, you can use XSU to insert the resulting canonical XML. For a specified database table, the canonical XML form is given by one row of XML output from a `SELECT *` query against the table.

**Syntax**

The syntax for this action is as follows, where `table_or_view_name` is a relative or absolute URL of an XSQL page to be included:

```xml
<xsql:insert-param table="table_or_view_name" name="string"/>
```

**Attributes**

Table 30–9 lists the optional attributes that you can use on the `<xsql:insert-param>` action.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name=&quot;string&quot;</td>
<td>Name of the parameter whose value contains XML to be inserted.</td>
</tr>
<tr>
<td>table=&quot;string&quot;</td>
<td>Name of the table, view, or synonym to use for inserting the XML data.</td>
</tr>
<tr>
<td>transform = &quot;URL&quot;</td>
<td>Relative or absolute URL of the XSLT transformation to use to transform the document to be inserted into canonical ROWSET/ROW format.</td>
</tr>
<tr>
<td>columns = &quot;string&quot;</td>
<td>Space-delimited or comma-delimited list of one or more column names whose values will be inserted. If supplied, then only these columns will be inserted. If not supplied, all columns will be inserted, with NULL values for columns whose values do not appear in the XML document.</td>
</tr>
<tr>
<td>commit = &quot;boolean&quot;</td>
<td>If set to <code>yes</code>, calls commit on the current connection after a successful execution of the insert. Valid values are <code>yes</code> (default) and <code>no</code>.</td>
</tr>
<tr>
<td>commit-batch-size = &quot;integer&quot;</td>
<td>If a positive, nonzero number integer is specified, then after each batch of integer inserted records, the XSQL processor issues a COMMIT. Default batch size is zero (0), which instructs the processor not to commit interim batches.</td>
</tr>
<tr>
<td>date-format = &quot;string&quot;</td>
<td>Date format mask to use for interpreting date field values in XML being inserted. Valid values are those for the java.text.SimpleDateFormat class.</td>
</tr>
<tr>
<td>error-param = &quot;string&quot;</td>
<td>Name of a page-private parameter that must be set to Error if a non-fatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
</tbody>
</table>

**Examples**

*Example 30–17* parses and transforms the contents of the HTML form parameter `xmlfield` for database insert.

*Example 30–17* Inserting XML Contained in an HTML Form Parameter

```xml
<?xml version="1.0"?>
<xsql:insert-param name="xmlfield" table="image_metadata_table"
```
transform='field-to-rowset.xsl' connection='demo' xmlns:xsql='urn:oracle-xsql'/>
<xsql:insert-request>

Purpose

Accepts data posted from an XML document or HTML form and uses the **XML SQL Utility (XSU)** to insert the content of an XML document in canonical form into a target table or view.

If an HTML Form has been posted, then the posted XML document is materialized from HTTP request parameters, cookies, and session variables. The XML document has the following form:

```xml
<request>
<parameters>
  <param1>value1</param1>
  ...
  <paramN>valueN</paramN>
</parameters>
</request>
```

By combining XSU with XSLT, you can transform XML into the canonical format expected by a given table. The XSQL engine uses XSU to insert the resulting canonical XML. For a specified database table, the canonical XML form is given by one row of XML output from a **SELECT *** query against the table.

Usage Notes

If you target a database view with an **INSERT**, then you can create **INSTEAD OF INSERT** triggers on the view to further automate the handling of the posted data. For example, an **INSTEAD OF INSERT** trigger on a view can use PL/SQL to check for the existence of a record and intelligently choose whether to do an **INSERT** or an **UPDATE** depending on the result.

Syntax

The syntax for this action is as follows:

```xml
<xsql:insert-request table="table"/>
```

Attributes

Table 30–10 lists the optional attributes that you can use on the `<xsql:insert-request>` action.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>table = &quot;string&quot;</strong></td>
<td>Name of the table, view, or synonym to use for inserting the XML data.</td>
</tr>
<tr>
<td><strong>transform = &quot;URL&quot;</strong></td>
<td>Relative or absolute URL of the XSLT transformation to use to transform the document to be inserted into canonical ROWSET/ROW format.</td>
</tr>
<tr>
<td><strong>columns = &quot;string&quot;</strong></td>
<td>Relative or absolute URL of the XSLT transformation to use to transform the document to be inserted into canonical ROWSET/ROW format.</td>
</tr>
<tr>
<td><strong>commit = &quot;boolean&quot;</strong></td>
<td>If set to yes (default), calls <strong>COMMIT</strong> on the current connection after a successful execution of the insert. Valid values are yes and no.</td>
</tr>
</tbody>
</table>
Example 30–18 parses and transforms the contents of the posted XML document or HTML Form for insert.

**Example 30–18  Inserting XML Received in a Parameter**

```xml
<?xml version='1.0'?>
<xs1:insert-request
  table='purchase_order'
  transform='purchaseorder-to-rowset.xsl'
  connection='demo'
  xmlns:xs1='urn:oracle-xsql'/>
```

---

### Table 30–10  (Cont.) Attributes for `<xs1:insert-request>`

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit-batch-size =</td>
<td>If a positive, nonzero number integer is specified, then after each batch of integer inserted records, the processor issues a COMMIT. The default batch size is zero (0) if not specified, which means that the processor does not commit interim batches.</td>
</tr>
<tr>
<td>&quot;integer&quot;</td>
<td></td>
</tr>
<tr>
<td>date-format = 'string'</td>
<td>Date format mask to use for interpreting date field values in XML being inserted. Valid values are those documented for the java.text.SimpleDateFormat class.</td>
</tr>
<tr>
<td>error-param = 'string'</td>
<td>Name of a page-private parameter that must be set to the string Error if a non-fatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
</tbody>
</table>
<xsql:query>

Purpose

Executes a SQL select statement and includes a canonical XML representation of the query result set in the data page. This action requires a database connection to be provided by supplying a connection="connname" attribute on the document element of the XSQL page in which it appears.

Syntax

The syntax for the action is the following:

```xml
<xsql:query>
  SELECT_Statement
</xsql:query>
```

Any legal SQL select statement is permissible as a substitution for the SELECT_Statement placeholder. If the select statement produces no rows, then you can provide a fallback query by including a nested <xsql:no-rows-query> element as follows:

```xml
<xsql:query>
  SELECT_Statement
  <xsql:no-rows-query>
    Fallback_SELECT_Statement
  </xsql:no-rows-query>
</xsql:query>
```

An <xsql:no-rows-query> element can itself contain nested <xsql:no-rows-query> elements to any level of nesting. The options available on the <xsql:no-rows-query> are identical to those legal on the <xsql:query> action element.

Attributes

The optional attributes listed in Table 30–11 can be supplied to control various aspects of the data retrieved and the XML produced by the <xsql:query> action.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bind-params = &quot;string&quot;</td>
<td>Ordered, space-delimited list of one or more XSQL parameter names. The values of these parameters are used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.</td>
</tr>
<tr>
<td>date-format = &quot;string&quot;</td>
<td>Date format mask to use for formatted date column and attribute values in the XML that is queried. Valid values are the same values legal for the java.text.SimpleDateFormat class.</td>
</tr>
<tr>
<td>error-param = &quot;string&quot;</td>
<td>Name of a page-private parameter that must be set to the string 'Error' if a nonfatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
<tr>
<td>error-statement = &quot;boolean&quot;</td>
<td>If set to no, suppresses the inclusion of the offending SQL statement in any <a href="">xsql:error</a> element generated. Valid values are yes (default) and no.</td>
</tr>
<tr>
<td>fetch-size = &quot;integer&quot;</td>
<td>Number of records to fetch in each round trip to the database. If not set, the default value is used as specified by the /XSQLConfig/processor/default-fetch-size configuration setting in XSQLConfig.xml.</td>
</tr>
<tr>
<td>id-attribute = &quot;string&quot;</td>
<td>XML attribute name to use instead of the default num for uniquely identifying each row in the result set. If the value is the empty string, then the row id attribute is suppressed.</td>
</tr>
</tbody>
</table>

30-26   Oracle XML Developer's Kit Programmer's Guide
Example 30–20 shows a simple XSQL page.

**Example 30–19  Hello World**

```xml
<?xml version='1.0'?>
<xsql:query connection="xmlbook" xmlns:xsql="urn:oracle-xsql">
    SELECT 'Hello, World!' AS text
    FROM DUAL
</xsql:query>
```

If you save Example 30–20 as hello.xsql and execute it in a browser, the XSQL page processor returns the following XML:

```xml
<?xml version = '1.0'?>
<ROWSET>
    <ROW num='1'>
        <TEXT>Hello, World!</TEXT>
    </ROW>
</ROWSET>
```

By default, the XML produced by a query reflects the column structure of its result set, with element names matching the names of the columns. Columns in the result with the following nested structure produce nested elements that reflect this structure:

- Object types
- Collection types
- CURSOR expressions

The result of a typical query containing different types of columns and returning one row might look like Example 30–20.
Example 30–20  Nested Structure Example

A <ROW> element repeats for each row in the result set. Your query can use standard SQL column aliasing to rename the columns in the result, which effectively renames the XML elements that are produced. Column aliasing is required for columns whose names otherwise are illegal names for an XML element.

For example, an <xsql:query> action as shown in Example 30–21 produces an error because the default column name for the calculated expression is an illegal XML element name.

Example 30–21  Query with Error

```xml
<?xml version="1.0"?>
<sql:query connection="demo" xmlns:sql="urn:oracle-xsql">
  SELECT TO_CHAR(hire_date,'DD-MON')
  FROM   employees
</sql:query>
```

You can fix the problem by using column aliasing as shown in Example 30–22.

Example 30–22  Query with Column Aliasing

```xml
<?xml version="1.0"?>
<sql:query connection="demo" xmlns:sql="urn:oracle-xsql">
  SELECT TO_CHAR(hire_date,'DD-MON') AS hiredate FROM employees
</sql:query>
```
**Purpose**

Executes an arbitrary stored function returning a `REF CURSOR` and includes the query result set in canonical XML format. This action requires a database connection to be provided by supplying a `connection="connname"` attribute on the document element of the XSQL page in which it appears.

Use this tag to invoke a stored procedure that determines what the query is and returns a cursor to the query. Used in this way, this tag also provides a weak level of security because it can hide the query from direct inspection.

**Syntax**

The syntax of the action is as follows, where `SCHEMA_NAME` represents an optional database schema name, `PACKAGE_NAME` represents an optional PL/SQL package name, and `FUNCTION_NAME` (required) specifies the name of a PL/SQL function:

```xml
<xsql:ref-cursor-function>
  [SCHEMA_NAME.]|[PACKAGE_NAME.]|FUNCTION_NAME(args);
</xsql:ref-cursor-function>
```

**Attributes**

The optional attributes are the same as for the `<xsql:query>` action listed in Table 30–11 except that `fetch-size` is not available for `<xsql:ref-cursor-function>`.

**Examples**

By exploiting dynamic SQL in PL/SQL, a function can conditionally construct a dynamic query before a cursor handle to its result set is returned to the XSQL page processor. The return value of the function must be of type `REF CURSOR`. Consider the PL/SQL package shown in Example 30–23.

**Example 30–23  DynCursor PL/SQL Package**

```sql
CREATE OR REPLACE PACKAGE DynCursor IS
  TYPE ref_cursor IS REF CURSOR;
  FUNCTION DynamicQuery(id NUMBER) RETURN ref_cursor;
END;
CREATE OR REPLACE PACKAGE BODY DynCursor IS
  FUNCTION DynamicQuery(id NUMBER) RETURN ref_cursor IS
    the_cursor ref_cursor;
    BEGIN
      IF id = 1 THEN -- Conditionally return a dynamic query as a REF CURSOR
        OPEN the_cursor -- An employees Query
        FOR 'SELECT employee_id, email FROM employees';
      ELSE
        OPEN the_cursor -- A departments Query
        FOR 'SELECT department_name, department_id FROM departments';
      END IF;
      RETURN the_cursor;
    END;
END;
END;
```

An `<xsql:ref-cursor-function>` can include the dynamic results of the `REF CURSOR` returned by this function as shown in Example 30–24.
Example 30–24  Executing a REF CURSOR Function

<?xml version="1.0"?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<xsql:ref-cursor-function connection='demo' xmlns:xsql='urn:oracle-xsql'>
   DynCursor.DynamicQuery();
</xsql:ref-cursor-function>
<xsql:set-cookie>

Purpose

Sets an HTTP cookie to a value. By default, the value remains for the lifetime of the current browser, but you can change its lifetime by supplying the optional max-age attribute. The value to be assigned to the cookie can be supplied by a combination of static text and other parameter values, or from the result of a SQL SELECT statement.

Because this feature is specific to the HTTP protocol, this action is only effective if the XSQL page in which it appears is processed by the XSQL servlet. If this action is encountered in an XSQL page processed by the XSQL command-line utility or the XSQLRequest programmatic API, then it does nothing.

Usage Notes

If you use the SQL statement option, then a single row is fetched from the result set and the parameter is assigned the value of the first column. This use requires a database connection to be provided by supplying a connection="connname" attribute on the document element of the XSQL page in which it appears.

If you need to set several cookie values based on the results of a single SQL statement, then do not use the name attribute. Instead, you can use the names attribute and supply a space-or-comma-delimited list of one or more cookie names.

Syntax

The syntax for this action is as follows, where paramname is the name of a parameter:

\[<xsql:set-cookie name="paramname" value="value"/>\]

Alternatively, you can use the following syntax, where SQL_statement is a SQL SELECT statement and paramname is the name of a parameter:

\[<xsql:set-cookie name="paramname">\n  SQL_statement\n</xsql:set-cookie>\]

Either the name or the names attribute is required. The value attribute and the contained SQL statement are mutually exclusive. The number of columns in the select list must match the number of cookies being set or an error message results.

Attributes

Table 30–12 lists the attributes supported by this action. Attributes in bold are required; all others are optional.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = &quot;string&quot;</td>
<td>Name of the cookie whose value you want to set. You must use name or names but not both.</td>
</tr>
<tr>
<td>names = &quot;string string ...&quot;</td>
<td>Space-or-comma-delimited list of the cookie names whose values you want to set. You must use name or names but not both.</td>
</tr>
<tr>
<td>bind-params = &quot;string&quot;</td>
<td>Ordered, space-delimited list of one or more XSQL parameter names. Values are used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.</td>
</tr>
</tbody>
</table>
Table 30–12 (Cont.) Attributes for <xsql:set-cookie>

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain = &quot;string&quot;</td>
<td>Domain in which cookie value is valid and readable. If domain is not set explicitly, it defaults to the fully-qualified host name (for example, server.biz.com) of the document creating the cookie.</td>
</tr>
<tr>
<td>error-param = &quot;string&quot;</td>
<td>Name of a page-private parameter that is set to the string 'Error' if a non-fatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
<tr>
<td>ignore-empty-value = &quot;boolean&quot;</td>
<td>Indicates whether the cookie assignment is ignored if the value to which it is being assigned is an empty string. Valid values are yes and no (default).</td>
</tr>
<tr>
<td>immediate = &quot;boolean&quot;</td>
<td>Indicates whether the cookie assignment is immediately visible to the current page. Typically, cookies set in the current request are not visible until the browser sends them back to the server in a subsequent request. Valid values are yes and no (default).</td>
</tr>
<tr>
<td>max-age = &quot;integer&quot;</td>
<td>Sets the maximum age of the cookie in seconds. Default is to set the cookie to expire when users current browser session terminates.</td>
</tr>
<tr>
<td>only-if-unset = &quot;boolean&quot;</td>
<td>Indicates whether the cookie assignment only occurs when the cookie currently does not exists. Valid values are yes and no (default).</td>
</tr>
<tr>
<td>path = &quot;string&quot;</td>
<td>Relative URL path within domain in which cookie value is valid and readable. If path is not set explicitly, then it defaults to the URL path of the document creating the cookie.</td>
</tr>
<tr>
<td>value = &quot;string&quot;</td>
<td>Sets the value to assign to the cookie.</td>
</tr>
</tbody>
</table>

Examples

Example 30–25 sets the HTTP cookie to the value of the parameter named choice.

Example 30–25 Setting a Cookie to a Parameter Value

```xml
<?xml version="1.0"?>
<xsql:set-cookie name="last_selection" value="{$choice}" xmlns:xsql="urn:oracle-xsql"/>
```

Example 30–5 sets the HTTP cookie to a value selected from the database.

Example 30–26 Setting a Cookie to a Database-Generated Value

```xml
<?xml version="1.0"?>
<xsql:set-cookie name="shopping_cart_id" bind-params="user"
    connection="demo" xmlns:xsql="urn:oracle-xsql">
    SELECT cartmgr.new_cart_id(UPPER(?)) FROM DUAL
</xsql:set-cookie>
```

Example 30–6 sets three cookies based on the result of a single SELECT statement.

Example 30–27 Setting Three Cookies

```xml
<?xml version="1.0"?>
<xsql:set-cookie names="paramname1 paramname2 paramname3"
    connection="demo" xmlns:xsql="urn:oracle-xsql">
    SELECT expression_or_column1, expression_or_column2, expression_or_column3
    FROM table
    WHERE clause_identifying_a_single_row
</xsql:set-cookie>
```
<xsql:set-page-param>

Purpose

Sets a page-private parameter to a value. The value can be supplied by a combination of static text and other parameter values, or alternatively from the result of a SQL SELECT statement.

Usage Notes

If you use the SQL statement option, then the program fetches a single row from the result set and assigns the parameter the value of the first column. This usage requires a database connection to be provided by supplying a connection="connname" attribute on the document element of the XSQL page in which it appears.

As an alternative to providing the value attribute, or a SQL statement, you can supply the xpath attribute to set the page-level parameter to the value of an XPath expression. The XPath expression is evaluated against an XML document or HTML form that has been posted to the XSQL pages processor. The value of the xpath attribute can be any valid XPath expression, optionally built using XSQL parameters as part of the attribute value like any other XSQL action element.

After a page-private parameter is set, subsequent action handlers can use this value as a lexical parameter, for example {@po_id}. Alternatively, action handlers can use this value as a SQL bind parameter value; they can reference its name in the bind-params attribute of any action handler that supports SQL operations.

If you need to set multiple session parameter values based on the results of a single SQL statement, instead of using the name attribute, then you can use the names attribute. You can supply a list, delimited by spaces or commas, of one or more session parameter names.

Syntax

The syntax for this action is as follows, where paramname is the name of a parameter and value is a value:

```xml
<xsql:set-page-param name="paramname" value="value"/>
```

Alternatively, you can use the following syntax, where SQL_statement is a SQL SELECT statement and paramname is the name of a parameter:

```xml
<xsql:set-page-param nname="paramname">
  SQL_statement
</xsql:set-page-param>
```

Alternatively, you can use the following syntax, where paramname is the name of a parameter and where expression is an XPath expression:

```xml
<xsql:set-page-param name="paramname" xpath="expression"/>
```

Either the name or the names attribute is required. The value attribute and the contained SQL statement are mutually exclusive.
Attributes

Table 30–13 lists the attributes supported by this action. Attributes in bold are required; all others are optional.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = &quot;string&quot;</td>
<td>Name of the page-private parameter whose value you want to set.</td>
</tr>
<tr>
<td>names = &quot;string string ...&quot;</td>
<td>Space-or-comma-delimited list of the page parameter names whose values you want to set. Either use the name or the names attribute, but not both.</td>
</tr>
<tr>
<td>bind-params = &quot;string&quot;</td>
<td>Ordered, space-delimited list of one or more XSQL parameter names. The values of these parameters are used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.</td>
</tr>
<tr>
<td>error-param = &quot;string&quot;</td>
<td>Name of a page-private parameter that must be set to the string 'Error' if a non-fatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
<tr>
<td>ignore-empty-value = &quot;boolean&quot;</td>
<td>Indicates whether the page-level parameter assignment is ignored if the value to which it is being assigned is an empty string. Valid values are yes and no (default).</td>
</tr>
<tr>
<td>quote-array-values = &quot;boolean&quot;</td>
<td>If the parameter name is a simple-valued parameter name (for example, myparam) and if treat-list-as-array=&quot;yes&quot; is specified, then specifying quote-array-values=&quot;yes&quot; will surround each string token with single quotes before separating the values with commas. Valid values are yes and no (default).</td>
</tr>
<tr>
<td>treat-list-as-array = &quot;boolean&quot;</td>
<td>Indicates whether the string-value assigned to the parameter is tokenized into an array of separate values before assignment. If any comma is present in the string, then the comma is used for separating tokens. Otherwise, spaces are used. Valid values are yes and no. The default value is yes if the parameter name being set is an array parameter name (for example, myparam[]), and default is no if the parameter name being set is a simple-valued parameter name like myparam.</td>
</tr>
<tr>
<td>value = &quot;string&quot;</td>
<td>Sets the value to assign to the parameter.</td>
</tr>
<tr>
<td>xpath = &quot;*XPathExpression&quot;</td>
<td>Sets the value of the parameter to an XPath expression evaluated against an XML document or HTML form that has been posted to the XSQL pages processor.</td>
</tr>
</tbody>
</table>

Examples

Example 30–28 sets multiple parameter values based on the results of a single SQL statement.

Example 30–28 Setting Multiple Page Parameters

```xml
<?xml version="1.0"?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<xsql:set-page-param names='paramname1 paramname2 paramname3' connection='demo' xmlns:xsql='urn:oracle-xsql'>
  SELECT expression_or_column1, expression_or_column2, expression_or_column3
  FROM table
  WHERE clause_identifying_a_single_row
</xsql:set-page-param>
```
Example 30–29 sets the page-level parameter to a value selected from database and then uses it as the value of an xsql:query attribute.

Example 30–29  Setting a Parameter to a Database-Generated Value

```xml
<?xml version='1.0'?><page connection='demo' xmlns:xsql='urn:oracle-xsql'>
  <xsql:set-page-param name='max-rows-pref'>
    SELECT max_rows
    FROM user_profile
    WHERE userid = '{@userid}'
  </xsql:set-page-param>
  <xsql:query max-rows='{@max-rows-pref}'>
    SELECT title, url
    FROM newsstory
    ORDER BY date_entered DESC
  </xsql:query>
</page>
```
<xsql:set-session-param>

Purpose

Sets an HTTP session-level parameter to a value. The value of the session-level parameter remains for the lifetime HTTP session of the current browser user. The session is controlled by the Web server. The value can be supplied by a combination of static text and other parameter values or from the result of a SQL SELECT statement.

Because this feature is specific to Java servlets, this action is only effective if the XSQL page in which it appears is processed by the XSQL servlet. If this action occurs in an XSQL page processed by the XSQL command-line utility or the XSQLRequest programmatic API, then it does nothing.

Usage Notes

If you use the SQL statement option, the XSQL processor fetches a single row from the result set and assigns the parameter the value of the first column. This use requires a database connection to be provided by supplying a connection="*connname*" attribute on the document element of the XSQL page in which it appears.

To set several session parameter values based on the results of a single SQL statement, do not use the name attribute. Instead, use the names attribute and supply a space-or-comma-delimited list of one or more session parameter names.

Syntax

The syntax for this action is as follows, where paramname is the name of a parameter and where value is a value:

```xml
<xsql:set-session-param name="paramname" value="value"/>
```

Alternatively, you can use the following syntax, where SQL_statement is a SQL SELECT statement and paramname is the name of a parameter:

```xml
<xsql:set-session-param name="paramname"> SQL_statement </xsql:set-session-param>
```

Either the name or the names attribute is required. The value attribute and the contained SQL statement are mutually exclusive.

Attributes

Table 30–14 lists the optional attributes supported by this action. Attributes in bold are required; all others are optional.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = &quot;string&quot;</td>
<td>Name of the session-level variable whose value you want to set. Either use the name or the names attribute, but not both.</td>
</tr>
<tr>
<td>names = &quot;string string ...&quot;</td>
<td>Space-or-comma-delimited list of the session parameter names whose values you want to set. Either use the name or the names attribute, but not both.</td>
</tr>
<tr>
<td>bind-params = &quot;string&quot;</td>
<td>Ordered, space-delimited list of one or more XSQL parameter names. The parameter values are used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.</td>
</tr>
</tbody>
</table>
**Example 30–30** sets multiple session parameter values based on the results of a single SELECT statement.

**Example 30–30  Setting Session Parameters**

```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
  <xsql:set-session-param names="paramname1 paramname2 paramname3">
    SELECT expression_or_column1, expression_or_column2, expression_or_column3
    FROM   table
    WHERE  clause_identifying_a_single_row
  </xsql:set-session-param>
  <!-- ... -->
</page>
```
<xsql:set-stylesheet-param>

Purpose

Sets a top-level XSLT stylesheet parameter to a value. The value can be supplied by a combination of static text and other parameter values, or from the result of a SQL SELECT statement. The stylesheet parameter will be set on any stylesheet used during the processing of the current page.

Usage Notes

If you use the SQL statement option, then a single row is fetched from the result set and the parameter is assigned the value of the first column. This use requires a database connection to be provided by supplying a connection="conname" attribute on the document element of the XSQL page in which it appears.

To set several stylesheet parameter values based on the results of a single SQL statement, do not use the name attribute. You can use the names attribute and supply a space-or-comma-delimited list of one or more stylesheet parameter names.

Syntax

The syntax for this action is as follows, where paramname is the name of a parameter and where value is a value:

<xsql:set-stylesheet-param name="paramname" value="value"/>

Alternatively, you can use the following syntax, where SQL_statement is a SQL SELECT statement and paramname is the name of a parameter:

<xsql:set-stylesheet-param name="paramname">
SQL_statement
</xsql:set-stylesheet-param>

Either the name or the names attribute is required. The value attribute and the contained SQL statement are mutually exclusive.

Attributes

Table 30–15 lists the optional attributes supported by this action. Attributes in bold are required; all others are optional.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = &quot;string&quot;</td>
<td>Name of the top-level stylesheet parameter whose value you want to set.</td>
</tr>
<tr>
<td>names = &quot;string string ...&quot;</td>
<td>Space-or-comma-delimited list of the top-level stylesheet parameter names whose values you want to set. Use the name or the names attribute, but not both.</td>
</tr>
<tr>
<td>bind-params = &quot;string&quot;</td>
<td>Ordered, space-delimited list of one or more XSQL parameter names. Parameter values are used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.</td>
</tr>
<tr>
<td>error-param = &quot;string&quot;</td>
<td>Name of a page-private parameter that has to be set to the string 'Error' if a non-fatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
<tr>
<td>ignore-empty-value = &quot;boolean&quot;</td>
<td>Indicates whether the stylesheet parameter assignment is to be ignored if the value to which it is being assigned is an empty string. Valid values are yes and no (default).</td>
</tr>
<tr>
<td>value = &quot;string&quot;</td>
<td>Sets the value to assign to the parameter.</td>
</tr>
</tbody>
</table>
Examples

Example 30–31 associate a stylesheet and uses the `<xsql:set-stylesheet-param>` action element to assign the value of the XSQL page parameter named `p_table` to the XSLT top-level stylesheet parameter named `table`.

Example 30–31 Setting a Stylesheet Parameter

```xml
<?xml version='1.0'?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<page connname='xmlbook' connection='{@p_connname}'>
  <xsql:query null-indicator='yes' xmlns:xsql='urn:oracle-xsql'>
    <![CDATA[
      SELECT *
      FROM `{@p_table}`
      WHERE rownum < 2
    ]]>}
  </xsql:query>
  <xsql:set-stylesheet-param name='table' value='{@p_table}'
    xmlns:xsql='urn:oracle-xsql' />
</page>
```
**Purpose**

Accepts data posted from an XML document or HTML form and uses the *XML SQL Utility (XSU)* to update the content of an XML document in canonical form from a target table or view.

By combining XSU with XSLT, you can transform XML into the canonical format expected by a given table. Afterward, you can use XSU to update the resulting canonical XML. For a specified database table, the canonical XML form is given by one row of XML output from a \( \text{SELECT} * \) query against the table.

**Syntax**

The syntax for this action is as follows:

```xml
<xsql:update-request table="table_name"/>
```

**Attributes**

Table 30–3 lists the attributes that you can use on the \( \text{<xsql:update-request>} \) action. Required attributes are in bold.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table = &quot;string&quot;</td>
<td>Name of the table, view, or synonym to use for updating the XML data.</td>
</tr>
<tr>
<td>key_columns = &quot;string string ...&quot;</td>
<td>Space-delimited or comma-delimited list of one or more column names. The processor uses the values of these names in the posted XML document to identify the existing rows to update.</td>
</tr>
<tr>
<td>transform = &quot;URL&quot;</td>
<td>Relative or absolute URL of the XSLT transformation to use to transform the document to be updated into canonical ROWSET/ROW format.</td>
</tr>
<tr>
<td>columns = &quot;string&quot;</td>
<td>Relative or absolute URL of the XSLT transformation to use to transform the document to be updated into canonical ROWSET/ROW format.</td>
</tr>
<tr>
<td>commit = &quot;boolean&quot;</td>
<td>If set to \textit{yes} (default), calls COMMIT on the current connection after a successful execution of the update. Valid values are \textit{yes} and \textit{no}.</td>
</tr>
<tr>
<td>commit-batch-size = &quot;integer&quot;</td>
<td>If a positive, nonzero \textit{integer} is specified, then after each batch of \textit{integer} updated records, the processor issues \texttt{COMMIT}. The default batch size is zero (0) if not specified, which means that the processor does not commit interim batches.</td>
</tr>
<tr>
<td>date-format = &quot;string&quot;</td>
<td>Date format mask to use for interpreting date field values in XML being updated. Valid values are those for the \texttt{java.text.SimpleDateFormat} class.</td>
</tr>
<tr>
<td>error-param = &quot;string&quot;</td>
<td>Name of a page-private parameter that must be set to Error if a nonfatal error occurs while processing this action. Valid value is any parameter name.</td>
</tr>
</tbody>
</table>

**Examples**

**Example 30–32** parses and transforms the contents of the posted XML document or HTML Form for update.

```xml
<?xml version="1.0"?>
<xsql:update-request table="purchase_order" key-columns="department_id"
    connection="demo" transforms="doc-to-departments.xsl"
    xmlns:xsql="urn:oracle-xsql"/>
```
This appendix contains the following topics:
- XML Standards Supported by the XDK
- Character Sets Supported by the XDK

XML Standards Supported by the XDK

This section contains the following topics:
- Summary of XML Standards Supported by the XDK
- XML Standards for the XDK for Java

Summary of XML Standards Supported by the XDK

Table 31–1 summarizes the standards supported by the XDK Components.

<table>
<thead>
<tr>
<th>Standard</th>
<th>Java</th>
<th>C</th>
<th>C++</th>
<th>Specification URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM 1.0</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.w3.org/TR/DOM-Level-1">http://www.w3.org/TR/DOM-Level-1</a></td>
</tr>
<tr>
<td>DOM 2.0 Core</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.w3.org/TR/DOM-Level-2-Core">http://www.w3.org/TR/DOM-Level-2-Core</a></td>
</tr>
<tr>
<td>DOM 2.0 Events</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.w3.org/TR/DOM-Level-2-Events">http://www.w3.org/TR/DOM-Level-2-Events</a></td>
</tr>
<tr>
<td>DOM 2.0 Transversal and Range</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.w3.org/TR/DOM-Level-2-Traversals-and-Range">http://www.w3.org/TR/DOM-Level-2-Traversals-and-Range</a></td>
</tr>
<tr>
<td>DOM 3.0 Load and Save</td>
<td>Partial</td>
<td>None</td>
<td>None</td>
<td><a href="http://www.w3.org/TR/2003/CR-DOM-Level-3-LS-20031107">http://www.w3.org/TR/2003/CR-DOM-Level-3-LS-20031107</a></td>
</tr>
<tr>
<td>DOM 3.0 Validation</td>
<td>Full</td>
<td>None</td>
<td>None</td>
<td><a href="http://www.w3.org/TR/2003/CR-DOM-Level-3-Val-20030710">http://www.w3.org/TR/2003/CR-DOM-Level-3-Val-20030710</a></td>
</tr>
<tr>
<td>JAXP 1.1 (JSR Standard)</td>
<td>Full</td>
<td>N/A</td>
<td>N/A</td>
<td><a href="http://java.sun.com/xml/downloads/jaxp.html">http://java.sun.com/xml/downloads/jaxp.html</a></td>
</tr>
<tr>
<td>JAXP 1.2 (JSR Standard)</td>
<td>Full</td>
<td>N/A</td>
<td>N/A</td>
<td><a href="http://java.sun.com/xml/downloads/jaxp.html">http://java.sun.com/xml/downloads/jaxp.html</a></td>
</tr>
<tr>
<td>SAX 1.0</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.saxproject.org">http://www.saxproject.org</a></td>
</tr>
<tr>
<td>SAX 2.0 Core</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.saxproject.org">http://www.saxproject.org</a></td>
</tr>
<tr>
<td>SAX 2.0 Extension</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.saxproject.org">http://www.saxproject.org</a></td>
</tr>
<tr>
<td>XML 1.0 (Second Edition)</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.w3.org/TR/REC-xml">http://www.w3.org/TR/REC-xml</a></td>
</tr>
<tr>
<td>XML Base</td>
<td>Only in XSLT</td>
<td>None</td>
<td>None</td>
<td><a href="http://www.w3.org/TR/xmlbase">http://www.w3.org/TR/xmlbase</a></td>
</tr>
<tr>
<td>XML Namespaces 1.0</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.w3.org/TR/REC-xml-names">http://www.w3.org/TR/REC-xml-names</a></td>
</tr>
<tr>
<td>XML Pipeline Definition Language 1.0 (Note)</td>
<td>Partial</td>
<td>None</td>
<td>None</td>
<td><a href="http://www.w3.org/TR/REC-xml-pipeline">http://www.w3.org/TR/REC-xml-pipeline</a></td>
</tr>
</tbody>
</table>
XML Standards Supported by the XDK

Table 31–1  (Cont.) Summary of XML Standards Supported by the XDK

<table>
<thead>
<tr>
<th>Standard</th>
<th>Java</th>
<th>C</th>
<th>C++</th>
<th>Specification URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML Schema language 1.0</td>
<td>Full</td>
<td>Full*</td>
<td>Full*</td>
<td><a href="http://www.w3.org/TR/xmlschema-0">http://www.w3.org/TR/xmlschema-0</a></td>
</tr>
<tr>
<td>XPath 1.0</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.w3.org/TR/xpath">http://www.w3.org/TR/xpath</a></td>
</tr>
<tr>
<td>XPath 2.0 Language (working draft dated 04 April 2005)</td>
<td>Full</td>
<td>None</td>
<td>None</td>
<td><a href="http://www.w3.org/TR/2005/WD-xpath20-20050404">http://www.w3.org/TR/2005/WD-xpath20-20050404</a></td>
</tr>
<tr>
<td>XPath 2.0 Data Model (working draft dated 04 April 2005)</td>
<td>Full</td>
<td>None</td>
<td>None</td>
<td><a href="http://www.w3.org/TR/2005/WD-xpath-datamodel-20050404/">http://www.w3.org/TR/2005/WD-xpath-datamodel-20050404/</a></td>
</tr>
<tr>
<td>XQuery 1.0 and XPath 2.0 Functions and Operators (working draft dated 04 April 2005)</td>
<td>Full</td>
<td>None</td>
<td>None</td>
<td><a href="http://www.w3.org/TR/2005/WD-xpath-functions-20050404">http://www.w3.org/TR/2005/WD-xpath-functions-20050404</a></td>
</tr>
<tr>
<td>XSLT 1.0</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td><a href="http://www.w3.org/TR/xslt">http://www.w3.org/TR/xslt</a></td>
</tr>
<tr>
<td>XSLT 2.0 (working draft dated 04 April 2005)</td>
<td>Partial*</td>
<td>None</td>
<td>None</td>
<td><a href="http://www.w3.org/TR/2005/WD-xslt20-20050404/">http://www.w3.org/TR/2005/WD-xslt20-20050404/</a></td>
</tr>
</tbody>
</table>

1 “DOM Level 3 Load and Save” on page 31-3 describes the relationship between DOM 3.0 Core and Load and Save.
2 “DOM 3.0 Validation” on page 31-3 describes the relationship between DOM 3.0 Core and Validation.
3 “Pipeline Definition Language Standard for the XDK for Java” on page 31-5 describes the parts of the standard that are not supported.
4 The Schema processor fully supports the functionality stated in the specification plus “XML Schema 1.0 Specification Errata” as published on http://www.w3.org/2001/05/xmlschema-errata.
5 “XSLT Standard for the XDK for Java” on page 31-3 describes the parts of the XSLT standard that are not supported.

XML Standards for the XDK for Java

This section contains the following topics:

- DOM Standard for the XDK for Java
- XSLT Standard for the XDK for Java
- JAXB Standard for the XDK for Java
- Pipeline Definition Language Standard for the XDK for Java

DOM Standard for the XDK for Java

**Note:** In Oracle Database 10g Release 2, the Java XDK implements the candidate recommendation versions of DOM Level 3.0 Load and Save and Validation specifications. Oracle plans to produce a release or patch set that will include an implementation of DOM Level 3.0 Load and Save and Validation recommendations. In order to conform to the recommendations, Oracle may be forced to make changes that are not backward compatible. During this period Oracle does not guarantee backward compatibility with respect to our DOM Load and Save, and Validation implementation. After the Java XDK is updated to conform to the recommendations, standard Oracle policies with respect to backwards compatibility will apply to the Oracle DOM Load and Save, and Validation implementation.

The DOM APIs include support for candidate recommendations of DOM Level 3 Validation and DOM Level 3 Load and Save.
DOM Level 3 Load and Save  The DOM Level 3 Load and Save module enables software developers to load and save XML content inside conforming products. The DOM 3.0 Core interface DOMConfiguration is referred by DOM 3 Load and Save. Although DOM 3.0 Core is not supported, a limited implementation of this interface is available.

The charset-overrides-xml-encoding configuration parameter is not supported by LSParser. Optional settings of the following configuration parameters are not supported by LSParser:

- disallow-doctype (true)
- ignore-unknown-character-denormalizations (false)
- namespaces (false)
- supported-media-types-only (true)

The discard-default-content configuration parameter is not supported by LSSerializer. Optional settings of the following configuration parameters are not supported by LSSerializer:

- canonical-form (true)
- format.pretty-print (true)
- ignore-unknown-character-denormalizations (false)
- normalize-characters (true)

DOM 3.0 Validation  DOM 3.0 validation allows users to retrieve the metadata definitions from XML schemas, query the validity of DOM operations and validate the DOM documents or sub-trees against the XML schema.

Some DOM 3 Core functions referred by Validation are implemented, but Core itself is not supported. Specifically, NameList and DOMStringList in DOM Core are supported for validation purposes. Because validation is based on an XML schema, you need to convert a DTD to an XML schema before using these functions.

XSLT Standard for the XDK for Java

The XSLT processor adds support for the current working drafts of XSLT 2.0, XPath 2.0, and the shared XPath/XQuery data model.
Some features of these specifications are not supported in the current release:

- The Schema Import and Static Typing features are not supported, but we do support XML Schema built-in types specified by the XPath 2.0 Datamodel.
- The `schema-element` and `schema-attribute` nodetests are not supported.
- The XSLT instruction `xsl:number` uses XSLT 1.0 semantics and syntax.
- The `use-when` standard attribute is not supported.
- The processor does not honor the attribute of `required` on `xsl:param`.
- Tunnel parameters are not supported.
- Regular expression instructions are not supported in XSLT.
- The XPath 2.0 functions `fn:tokenize`, `fn:matches`, and `fn:replace` are not supported.
- `format-dateTime`, `format-date`, and `format-time` functions are not supported.
- The content model for `xsl:attribute`, `xsl:comment`, `xsl:message` and the way to compute key values of `xsl:key` and `xsl:sort` are still 1.0 behavior.
- `attribute` `[xsl:import-namespace` for `xsl:copy`, `xsl:element`, and literal result elements is not supported.

Updates to the W3C specifications for XPath 2.0 and XSLT 2.0 resulted in certain differences in behavior from 10g Release 1. For 10g Release 1 compatible behavior set the system property `oracle.xdkjava.compatibility.version=10.1.0`.

XPath 2.0 - Differences between 10g Release 1 and 10g Release 2

1. `RangeExpr`, behavior for `(m to n)`, where `m > n` changed. Earlier we treated it as `(n to m)`, reverse sequence. As described in the April 2005 draft, we return an empty sequence.

2. `isnot` operator was removed as per the Apr 2005 draft.

3. `getEffectiveBooleanValue` definition (`fn:boolean`) updated as described in the April 2005 draft. Empty string value will result in exception (FORG006) instead of
returning false. All cases not handled by getEffectiveBooleanValue will result in an exception (FORG006). XPath 1.0 behavior for fn:boolean will remain the same.

XSLT 2.0 - Difference between 10g Release 1 and 10g Release 2

normalize-unicode has been changed to normalization-form, the allowed attribute values have been changed from "yes" | "no" to "NFC" | "NFD" | "NKFC" | "NKFD" | \x{2026}\x{2026} as described in the Nov. 2004 draft.

JAXB Standard for the XDK for Java

The Oracle Database XDK implementation of the JAXB specification does not support the following features:
- Javadoc generation
- XML Schema component "any" and substitution groups

Pipeline Definition Language Standard for the XDK for Java

The XML Pipeline processor differs from the W3C Note as follows:
- The parser processes DOMParserProcess and SAXParserProcess are included in the XML pipeline (Section 1).
- Only the final target output is checked to see if it is up-to-date with respect to the available pipeline inputs. The XML Pipeline processor does not determine whether the intermediate outputs of every process are up-to-date (Section 2.2).
- For the select attribute, anything in between double-quotes ("...") is considered to be a string literal.
- The XML Pipeline processor throws an error if more that one process produces the same infoset (Section 2.4.2.3).
- The <document> element is not supported (Section 2.4.2.8).

Character Sets Supported by the XDK

This section contains the following topics:
- Character Sets Supported by the XDK for Java
- Character Sets Supported by the XDK for C

Character Sets Supported by the XDK for Java

XML Schema processor for Java supports documents in the following encodings:
- BIG
- EBCDIC-CP-*
- EUC-JP
- EUC-KR
- GB2312
- ISO-2022-JP
- ISO-8859-1to -9
- ISO-10646-UCS-2
Character Sets Supported by the XDK

- ISO-10646-UCS-4
- KOI8-R
- Shift_JIS
- US-ASCII
- UTF-8
- UTF-16

Character Sets Supported by the XDK for C

The XDK parser for C supports over 300 IANA character sets. These character sets include the following:

- UTF-8
- UTF-16
- UTF16-BE
- UTF16-LE
- US-ASCII
- ISO-10646-UCS-2
- ISO-8859-{1-9, 13-15}
- EUC-JP
- SHIFT_JIS
- BIG5
- GB2312
- GB_2312-80
- HZ-GT-2312
- KOI8-R
- KSC5601
- EUC-KR
- ISO-2022-CN
- ISO-2022-JP
- WINDOWS-{1250-1258}
- EBCDIC-CP-{US, CA, NL, WT, DK, NO, FI, SE, IT, ES, GB, FR, HE, BE, CH, ROECE, YU, IS, AR}
- IBM{037, 273, 277, 278, 280, 284, 285, 297, 420, 424, 437, 500, 775, 850, 852, 855, 857, 858, 860, 861, 863, 865, 866, 869, 870, 871, 1026, 01140, 01141, 01142, 01143, 01144, 01145, 01146, 01147, 01148}

You can use any alias of the preceding character sets. In addition, you can use any character set specified in Appendix A, Character Sets, of the Oracle Database Globalization Support Guide with the exception of IW7IS960.
This section lists error messages that may be encountered in applications that use Oracle XDK for Java. These are divided into these principal sections:

- **XML Error Messages**
- **TXU Error Messages**
- **XSU Error Messages**

See Also: [http://www.w3.org/TR/xquery/#id-errors](http://www.w3.org/TR/xquery/#id-errors) for the XQuery error messages

### XML Error Messages

These error messages may occur during the execution of XML interfaces.

### XML Parser Error Messages

These error messages are in the range XML-20000 through XML-20999.

**XML-20003: missing token string at line string, column string**

An expected token was not found in the input data.

**Action:** Check/update the input data to fix the syntax error.

**XML-20004: missing keyword string at line string, column string**

**Cause:** An expected keyword was not found in the input data.

**Action:** Check/update the input data to the correct keyword.

**XML-20005: missing keyword string or string at line string, column string**

**Cause:** An expected keyword was not found in the input data.

**Action:** Check/update the input data to the correct keyword.

**XML-20006: unexpected text at line string, column string; expected EOF**

**Cause:** More text was found after the end-tag of the root element.

**Action:** The end-tag of the root element can be followed only by comments, PI, or white space. Remove the extra text after the end-tag.
XML-20007: missing content model in element declaration at line string, column string

Cause: The element declaration was missing the required content model spec. See Production [45] in XML 1.0 2nd Edition.

Action: Add the required content spec to the element declaration.

XML-20008: missing element name in content model at line string, column string

Cause: The content model in the element declaration was invalid, the content particle requires an element name. See Production [48] in XML 1.0 2nd Edition.

Action: Add the element name to fix the content spec syntactically.

XML-20009: target name string of processing instruction at line string, column string is reserved


Action: If the PI is meant to be XML declaration, make sure the declaration occurs at the very beginning of the file. Otherwise, change to name of the PI.

XML-20010: missing notation name in unparsed entity declaration at line string, column string

Cause: The notation name used in the unparsed entity declaration did not match the name in a declared notation. See Production [76] in XML 1.0 2nd Edition.

Action: Add the notation declaration to the DTD.

XML-20011: missing attribute type in attribute-list declaration at line string, column string

Cause: The attribute type was missing the attribute-list declaration. One of the following types CDATA, ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, or NMTOKENS must be added. See Production [52], [53] in XML 1.0 2nd Edition.

Action: Check and correct attribute declaration.

XML-20012: missing white space at line string, column string

Cause: The required white space was missing.

Action: Add white space to fix the syntax error.

XML-20013: invalid character string in entity value at line string, column string

Cause: An invalid character was used in the entity value, the characters '&', '%', and (' or ' based on the value delimiters) are invalid. See Production [9] in XML 1.0 2nd Edition.

Action: Use entity or character references instead of the characters. For example, &amp; or &apos;38; can be used instead of ' &'

XML-20014: -- not allowed in comment at line string, column string


Action: Fix the comment, and use "--" only as part of end of comment " -->"
XML-20015: "]> not allowed in text at line string, column string
Cause: "]>" is not allowed in text, it is used only as end marker for CDATA Section. See Production [14] in XML 1.0 2nd Edition.
Action: Fix the text content by using &gt; or char ref for '>'

XML-20016: white space not allowed before occurrence indicator at line string, column string
Cause: White space is not allowed in the contentspec before the occurrence indicator. For example, <!ELEMENT x (a,b)> is not valid. See Production [47], [48] in XML 1.0 2nd Edition.
Action: Fix the contentspec by removing the extra space

XML-20017: occurrence indicator string not allowed in mixed-content at line string, column string
Cause: Occurrence is not allowed in mixed content declaration. For example, <!ELEMENT x (#PCDATA)> is not valid. See Production [51] in XML 1.0 2nd Edition.
Action: Fix the syntax to remove the occurrence indicator.

XML-20018: content list not allowed inside mixed-content at line string, column string
Cause: Content list is not allowed in mixed-content declaration. For example, <!ELEMENT x (#PCDATA | (a,b))> is not valid. See Production [51] in XML 1.0 2nd Edition.
Action: Fix the syntax to remove the content list.

XML-20019: duplicate element string in mixed-content declaration at line string, column string
Cause: Duplicate element name was found in mixed-content declaration. For example, <!ELEMENT x (#PCDATA | a | a)> is not valid. See Production [51] in XML 1.0 2nd Edition.
Action: Remove the duplicate element name.

XML-20020: root element string does not match the DOCTYPE name string at line string, column string
Cause: failed: The Name in the document type declaration must match the element type of the root element. For example: <?xml version="1.0"?> <!DOCTYPE greeting [ <!ELEMENT greeting (#PCDATA)> ]> <salutation>Hello!</salutation> The document's root element, salutation, does not match the root element declared in the DTD (greeting).
Action: Correct the document.

XML-20021: duplicate element declaration string at line string, column string
Cause: Element was declared twice in the DTD.
Action: Remove the duplicate declaration.

XML-20022: element string has multiple ID attributes at line string, column string
Cause: failed: No element type may have more than one ID attribute specified.
Action: Correct the document, by removing the duplicate ID attribute declaration.
XML-20023: ID attribute *string* in element *string* must be #IMPLIED or #REQUIRED at line *string*, column *string*

**Cause:** failed: An ID attribute must have a declared default of #IMPLIED or #REQUIRED.

**Action:** Fix the attribute declaration.

XML-20024: missing required attribute *string* in element *string* at line *string*, column *string*

**Cause:** failed: If the default declaration is the keyword #REQUIRED, then the attribute must be specified for all elements of the type in the attribute-list declaration.

**Action:** Fix the input document by specifying the required attribute.

XML-20025: duplicate ID value: *string*

**Cause:** Values of type ID must match the Name production. A name must not appear more than once in an XML document as a value of this type; i.e., ID values must uniquely identify the elements which bear them.

**Action:** Fix the input document by removing the duplicate ID value.

XML-20026: undefined ID value *string* in IDREF

**Cause:** failed: "Values of type IDREF must match value of some ID attribute.

**Action:** Fix the document by adding an ID corresponding to the IDREF, or removing the IDREF.

XML-20027: attribute *string* in element *string* has invalid enumeration value *string* at line *string*, column *string*

**Cause:** failed: Values of this type must match one of the Nmtoken tokens in the declaration.

**Action:** Fix the attribute value to match one of the enumerated values.

XML-20028: attribute *string* in element *string* has invalid value *string*, must be *string* at line *string*, column {5}

**Cause:** failed: If an attribute has a default value declared with the #FIXED keyword, instances of that attribute must match the default value.

**Action:** Update the attribute value to match the fixed default value.

XML-20029: attribute default must be REQUIRED, IMPLIED, or FIXED at line *string*, column *string*

**Cause:** The declared default value must meet the lexical constraints of the declared attribute type.

**Action:** Use one of REQUIRED, IMPLIED, or FIXED for attribute default decl.

XML-20030: invalid text in content of element *string* at line *string*, column *string*

**Cause:** The element does not allow text in content. An element is valid if there is a declaration matching element decl where the Name matches the element type, and one of the following holds:
The declaration matches children and the sequence of child elements belongs to the language generated by the regular expression in the content model, with optional white space (characters matching the nonterminal S) between the start-tag and the first child element, between child elements, or between the last child element and the end-tag. Note that a CDATA section containing only white space does not match the non-terminal S, and hence cannot appear in these positions.

**Action:** Fix the content by removing unexpected text.

**XML-20031:** invalid element *string* in content of element *string* at line *string*, column *string*

**Cause:** The element has invalid content. An element is valid if there is a declaration matching element decl where the Name matches the element type, and one of the following holds:

1. The declaration matches children and the sequence of child elements belongs to the language generated by the regular expression in the content model, with optional white space (characters matching the non-terminal S) between the start-tag and the first child element, between child elements, or between the last child element and the end-tag. Note that a CDATA section containing only white space does not match the non-terminal S, and hence cannot appear in these positions.

2. The declaration matches Mixed and the content consists of character data and child elements whose types match names in the content model.

**Action:** Fix the content by removing unexpected elements.

**XML-20032:** incomplete content in element *string* at line *string*, column *string*

**Cause:** The element has invalid content. An element is valid if there is a declaration matching element decl where the Name matches the element type, and one of the following holds:

1. The declaration matches children and the sequence of child elements belongs to the language generated by the regular expression in the content model, with optional white space (characters matching the non-terminal S) between the start-tag and the first child element, between child elements, or between the last child element and the end-tag. Note that a CDATA section containing only white space does not match the non-terminal S, and hence cannot appear in these positions.

2. The declaration matches Mixed and the content consists of character data and child elements whose types match names in the content model.

**Action:** Fix the content by removing unexpected elements.

**XML-20033:** invalid replacement-text for entity *string* at line *string*, column *string*

**Cause:** Parameter-entity replacement text must be properly nested with markup declarations. That is to say, if either the first character or the last character of a markup declaration (markup decl above) is contained in the replacement text for a parameter-entity reference, both must be contained in the same replacement text.

**Action:** Fix the entity value.

**XML-20034:** end-element tag *string* does not match start-element tag *string* at line *string*, column *string*

**Cause:** The Name in an element's end-tag must match the element type in the start-tag.
**Action:** Fix the end-tag or start-tag to match the other.

**XML-20035: duplicate attribute string in element string at line string, column string**

**Cause:** No attribute name may appear more than once in the same start-tag or empty-element tag.

**Action:** Remove the duplicate attribute.

**XML-20036: invalid character string in attribute value at line string, column string**

**Cause:** An invalid character was used in the attribute value, the characters '&', '<' and (' or ' based on the value delimiters) are invalid. See Production [10] in XML 1.0 2nd Edition.

**Action:** Use entity or character references instead of the characters For example, `&amp;` or `&#38;` can be used instead of `&`

**XML-20037: invalid reference to external entity string in attribute string at line string, column string**

**Cause:** Attribute values cannot contain direct or indirect entity references to external entities.

**Action:** Fix document to remove reference to external entity in attribute.

**XML-20038: invalid reference to unparsed entity string in element string at line string, column string**

**Cause:** An entity reference must not contain the name of an unparsed entity. Unparsed entities may be referred to only in attribute values declared to be of type ENTITY or ENTITIES.

**Action:** Fix document to remove reference to unparsed entity in content.

**XML-20039: invalid attribute type string in attribute-list declaration at line string, column string**

**Cause:** Invalid attribute type was used in the attribute-list declaration. One of the following types CDATA, ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, or NMTOKENS must be added. See Production [52], [53] in XML 1.0 2nd Edition.

**Action:** Check and correct attribute declaration.

**XML-20040: invalid character string in element content at line string, column string**

**Cause:** Characters referred to using character references must match the production for Char.

**Action:** Fix the document by removing the invalid character or char-ref.

**XML-20041: entity reference string refers to itself at line string, column string**

**Cause:** A parsed entity must not contain a recursive reference to itself, either directly or indirectly.

**Action:** Fix the document.

**XML-20042: invalid Nmtoken: string**

**Cause:** Values of this type must match one of the Nmtoken tokens in the declaration, and must be valid Nmtoken"
Action: Fix the attribute value.

XML-20043: invalid character string in public identifier at line string, column string
Action: Fix the public identifier.

XML-20044: undeclared namespace prefix string used at line string, column string
Cause: The prefix was not defined in any namespace declaration in scope.
Action: Add a namespace declaration to define the prefix.

XML-20045: attribute string in element string must be an unparsed entity at line string, column string
Cause: Values of type ENTITY must match the Name production, values of type ENTITIES must match Names; each Name must match the name of an unparsed entity declared in the DTD.
Action: Fix the attribute value to refer to an unparsed entity.

XML-20046: undeclared notation string used in unparsed entity string at line string, column string
Cause: Values of this type must match one of the notation names included in the declaration; all notation names in the declaration must be declared.
Action: Fix the notation name in the unparsed entity declaration.

XML-20047: missing element declaration string
Cause: The element declaration referred to by an attribute declaration was not found in the DTD.
Action: Fix the DTD by adding the element declaration.

XML-20048: duplicate entity declaration string at line string, column string
Cause: Warning regarding duplicate entity declaration.
Action: No action required.

XML-20049: invalid use of NDATA in parameter entity declaration at line string, column string
Cause: NDATA declaration was found in parameter entity declaration. It is allowed only in general unparsed entity declaration. See Production [72], [74] in XML 1.0 2nd Edition.
Action: Fix the entity declaration.

XML-20050: duplicate attribute declaration string at line string, column string
Cause: Warning regarding duplicate attribute declaration.
Action: No action required.

XML-20051: duplicate notation declaration string at line string, column string
Cause: Only one notation declaration can declare a given Name.
Action: Fix the document by removing the duplicate notation.

XML-20052: undeclared attribute *string* used at line *string*, column *string*
Cause: The attribute declaration was not found in the DTD.
Action: Fix the DTD by adding the attribute declaration.

XML-20053: undeclared element *string* used at line *string*, column *string*
Cause: The element declaration was not found in the DTD.
Action: Fix the DTD by adding the element declaration.

XML-20054: undeclared entity *string* used at line *string*, column *string*
Cause: The entity declaration was not found in the DTD.
Action: Fix the DTD by adding the element declaration.

XML-20055: invalid document returned by NodeFactory's createDocument
Cause: The document returned by createDocument function of NodeFactory was invalid, either it was null or instance of an unsupported class.
Action: Fix NodeFactory implementation to return an instance of XMLDocument or its subclass.

XML-20056: invalid SAX feature *string*
Cause: The SAX feature supplied was not a valid feature name.
Action: Refer to documentation for a valid list of features.

XML-20057: invalid value *string* passed for SAX feature *string*
Cause: The value supplied for the SAX feature was not valid.
Action: Refer to documentation for a valid list of features and their corresponding values.

XML-20058: invalid SAX property *string*
Cause: The SAX property supplied was not a valid property name.
Action: Refer to documentation for a valid list of properties.

XML-20059: invalid value passed for SAX property *string*
Cause: The value supplied for the SAX property was not valid.
Action: Refer to documentation for a valid list of properties and their corresponding values.

XML-20060: Error occurred while opening URL *string*
Cause: An error occurred while opening the supplied URL.
Action: Verify the URL, and take appropriate action to allow data to be read.

XML-20061: invalid byte stream *string* in UTF8 encoded data
Cause: The input data contained bytes that are not valid w.r.t to UTF8 encoding scheme.
**XML-20062: 5-byte UTF8 encoding not supported**

**Cause:** The XML Parser does not support 5-byte UTF8 encoding scheme. It is also possible that invalid UTF8 characters were misinterpreted as 5-byte UTF8 encoding.

**Action:** If the data contains invalid UTF8 bytes, fix the input, otherwise if 5-byte UTF8 supported is required, please contact Oracle Support.

**XML-20063: 6-byte UTF8 encoding not supported**

**Cause:** The XML Parser does not support 6-byte UTF8 encoding scheme. It is also possible that invalid UTF8 characters were misinterpreted as 6-byte UTF8 encoding.

**Action:** If the data contains invalid UTF8 bytes, fix the input, otherwise if 6-byte UTF8 supported is required, please contact Oracle Support.

**XML-20064: invalid XML character string**

**Cause:** Invalid XML character was found in the input data.

**Action:** Fix the input data.

**XML-20065: encoding string doesn’t match encoding string in XML declaration**

**Cause:** The encoding of the data (either by auto-detection or user supplied) didn’t match the encoding specified in the XML declaration.

**Action:** Fix the XML declaration to match the encoding of the data.

**XML-20066: encoding string not supported**

**Cause:** The XML Parser does not support the specified encoding.

**Action:** If the support for the encoding is required, please contact Oracle Support.

**XML-20067: invalid InputSource returned by EntityResolver’s resolveEntity**

**Cause:** An invalid instance of InputSource was returned by the EntityResolver. An InputSource can be invalid if the none of Reader, InputStream, and SystemId were initialized or if the SystemId was invalid.

**Action:** Fix the EntityResolver class to return a valid instance of InputSource.

**XML-20100: Expected string.**

**XML-20101: Expected string or string.**

**XML-20102: Expected string, string, or string.**

**XML-20103: Illegal token in content model.**
XML-20104: Could not find element with ID string.

XML-20105: ENTITY type Attribute value string does not match any unparsed Entity.

XML-20106: Could not find Notation string.

XML-20107: Could not find declaration for element string.

XML-20108: Start of root element expected.

XML-20109: PI with the name 'xml' can occur only in the beginning of the document.

XML-20110: #PCDATA expected in mixed-content declaration.

XML-20111: Element string repeated in mixed-content declaration.

XML-20112: Error opening external DTD string.

XML-20113: Unable to open input source (string).

XML-20114: Bad conditional section start syntax, expected '['.

XML-20115: Expected ']]>' to end conditional section.

XML-20116: Entity string already defined, using the first definition.

XML-20117: NDATA not allowed in parameter entity declaration.

XML-20118: NDATA value required.

XML-20119: Entity Value should start with quote.
XML-20120: Entity value not well-formed.

XML-20121: End tag does not match start tag string.

XML-20122: '=' missing in attribute.

XML-20123: '>' Missing from end tag.

XML-20124: An attribute cannot appear more than once in the same start tag.

XML-20125: Attribute value should start with quote.

XML-20126: '<' cannot appear in attribute value.

XML-20127: Reference to an external entity not allowed in attribute value.

XML-20128: Reference to unparsed entity not allowed in element content.

XML-20129: Namespace prefix string used but not declared.

XML-20130: Root element name must match the DOCTYPE name.

XML-20131: Element string already declared.

XML-20132: Element cannot have more than one ID attribute.

XML-20133: Attr type missing.

XML-20134: ID attribute must be declared #IMPLIED or #REQUIRED.

XML-20135: Attribute string already defined, using the first definition.
XML-20136: Notation string already declared.

XML-20137: Attribute string used but not declared.

XML-20138: REQUIRED attribute string is not specified.

XML-20139: ID value string is not unique.

XML-20140: IDREF value string does not match any ID attribute value.

XML-20141: Attribute value string should be one of the declared enumerated values.

XML-20142: Unknown attribute type.

XML-20143: Unrecognized text at end of attribute value.

XML-20144: FIXED type Attribute value not equal to the default value string.

XML-20145: Unexpected text in content of Element string.

XML-20146: Unexpected text in content of Element string, expected elements string.

XML-20147: Invalid element string in content of string, expected closing tag.

XML-20148: Invalid element string in content of string, expected elements string.

XML-20149: Element string used but not declared.

XML-20150: Element string not complete, expected elements string.

XML-20151: Entity string used but not declared.
XML-20170: Invalid UTF8 encoding.

XML-20171: Invalid XML character(string).

XML-20172: 5-byte UTF8 encoding not supported.

XML-20173: 6-byte UTF8 encoding not supported.


XML-20190: Whitespace required.

XML-20191: '>' required to end DTD.

XML-20192: Unexpected text in DTD.

XML-20193: Unexpected EOF.

XML-20194: Unable to write to output stream.

XML-20195: Encoding not supported in PrintWriter.

XML-20200: Expected string instead of string.

XML-20201: Expected string instead of string.

XML-20202: Expected string to be string.

XML-20205: Expected string.

XML-20206: Expected string or string.
XML-20210: Unexpected *string*.

XML-20211: *string* is not allowed in *string*.

XML-20220: Invalid InputSource.

XML-20221: Invalid char in text.

XML-20230: Illegal change of encoding: from *string* to *string*.

XML-20231: Encoding *string* is not currently supported.

XML-20240: Unable to open InputSource.

XML-20241: Unable to open entity *string*.

XML-20242: Error opening external DTD *string*.

XML-20250: Missing entity *string*.

XML-20251: Cyclic Entity Reference in entity *string*.

XML-20280: Bad character (*string*).

XML-20281: NMToken must contain atleast one NMChar.

XML-20282: *string* not allowed in a PubIdLiteral.

XML-20284: Illegal white space before optional character in content model.

XML-20285: Illegal mixed content model.
XML-20286: Content list not allowed inside mixed content model.

XML-20287: Content particles not allowed inside mixed content model.

XML-20288: Invalid default declaration in attribute declaration.

XML-20500: SAX feature string not recognized.

XML-20501: SAX feature string not supported.

XML-20502: SAX property string not recognized.

XML-20503: SAX property string not supported.

DOM Error Messages

These error messages are in the range XML-21000 through XML-21999.

XML-21000: invalid size string specified
Cause: An invalid size or count was passed to a DOM function.
Action: Correct the argument passed to a valid value.

XML-21001: invalid index string specified; must be between 0 and string
Cause: An invalid index was passed to a DOM function.
Action: Correct the argument passed to a valid value specified by the bounds in the error message.

XML-21002: cannot add an ancestor as a child node
Cause: The DOM operation was trying to add an ancestor node as a child. This can lead to inconsistencies in the tree, so it is not allowed.
Action: Check the application to fix the usage.

XML-21003: node of type string cannot be added to node of type string
Cause: The DOM specification does not allow the parent-child combination used in the DOM operation.
Action: Refer to DOM specification to fix the usage.

XML-21004: document node can have only one string node as child
Cause: The XML well-formedness requires that the document node have only one element node as its child. The application tried adding a second element node.
**Action:** Fix usage in the application.

**XML-21005: node of type string cannot be added to attribute list**
**Cause:** The attribute list (instance of NamedNodeMap) can contain only attribute nodes.
**Action:** Fix usage of NamedNodeMap.

**XML-21006: cannot add a node belonging to a different document**
**Cause:** The node being added was created by a different document. The DOM specification does not allow use of nodes across documents.
**Action:** Use importNode or adoptNode to move a node from one document to another, before adding it.

**XML-21007: invalid character string in name**
**Cause:** The qualified or local name passed was invalid.
**Action:** Fix the name to contain only valid characters.

**XML-21008: cannot set value for node of type string**
**Cause:** The node of the specified type cannot have value.
**Action:** Fix usage of DOM functions.

**XML-21009: cannot modify descendants of entity or entity reference nodes**
**Cause:** The descendants of entity or entity reference nodes are read-only nodes, and modification is not allowed.
**Action:** Fix usage of DOM functions.

**XML-21010: cannot modify DTD's content**
**Cause:** DTD and all its content is read-only and cannot be modified.
**Action:** Fix usage of DOM functions.

**XML-21011: cannot remove attribute; not found in the current element**
**Cause:** An attempt was made to remove an attribute that does not belong to the current element.
**Action:** Fix usage in application.

**XML-21012: cannot remove or replace node; it is not a child of the current node**
**Cause:** An attempt was made to remove an node that does not belong to the current node as a child.
**Action:** Fix usage in application.

**XML-21013: parameter string not recognized**
**Cause:** The DOM parameter was not recognized.
**Action:** See documentation for a valid list of parameters.
XML-21014: value string of parameter string is not supported
Cause: The DOM parameter was not recognized.
Action: See documentation for a valid list of parameters.

XML-21015: cannot add attribute belonging to another element
Cause: An attempt was made to add an attribute that belonged to another element.
Action: Fix usage in application.

XML-21016: invalid namespace string for prefix string
Cause: The namespace for xml, and xmlns prefixes is fixed, and usage mustmatch these.
Action: Correct the namespace for the prefixes, namespaces are xml = http://www.w3.org/XML/1998/namespace xmlns = http://www.w3.org/2000/xmlns/

XML-21017: invalid qualified name: string
Cause: The qualified name passed to a DOM function was invalid.
Action: Fix the qualified name.

XML-21018: conflicting namespace declarations string and string for prefix string
Cause: The DOM tree has conflicting namespace declarations for the same prefix. Such a DOM tree cannot be serialized.
Action: Fix the DOM tree, before printing it.

XML-21019: string object is detached
Cause: The object was detached, no operations are supported on a detached object. The object can be a Range or iterator object
Action: Fix the usage in application.

XML-21020: bad boundary specified; cannot partially select a node of type string
Cause: The boundary specified in the range was invalid. The selection can be partial only for text nodes.
Action: Fix the usage in the application.

XML-21021: node of type string does not support range operation string
Cause: The range operation is not supported on the node type specified.
Action: Refer to DOM documentation for restrictions of node types for each range operation.

XML-21022: invalid event type: string
Cause: The event type passed was invalid.
Action: Fix usage in the application.

XML-21023: prefix not allowed on nodes of type string
Cause: The application tried to set prefix on a node on which prefix is not allowed
Action: Fix usage in the application.

**XML-21024: import not allowed on nodes of type string**  
Cause: The application tried to import a node of type DOCUMENT or DOCUMENT FRAGMENT.  
Action: Fix usage in the application.

**XML-21025: rename not allowed on nodes of type string**  
Cause: The application tried to import a node of type other than ELEMENT or ATTRIBUTE.  
Action: Fix usage in the application.

**XML-21026: Unrepresentable character in node: string**  
Cause: A node contains an invalid character, eg. CDATA section contain a termination character.  
Action: Set appropriate DOMConfiguration parameter.

**XML-21027: Namespace normalization error in node: string**  
Cause: Namespace fixup cannot be performed on this node.  
Action: Set namespace normalization to false.

**XML-21997: function not supported on THICK DOM**  
Cause: A function on THICK (for example, XDB based) DOM which is not supported was called.  
Action: Refer to the XDK documentation for possible alternatives for functions not supported on THICK DOM.

**XML-21998: system error occurred: string**  
Cause: Non-dom related system errors occurred.  
Action: Check with ORA error(s) embedded in the message and consult with developers for possible causes.

### XSL Transformation Error Messages

These error messages are in the range XML-22000 through XML-22999.

**XML-22000: Error while parsing XSL file (string).**

**XML-22001: XSL Stylesheet does not belong to XSLT namespace.**

**XML-22002: Error while processing include XSL file (string).**

**XML-22003: Unable to write to output stream (string).**
XML-22004: Error while parsing input XML document (string).

XML-22005: Error while reading input XML stream (string).

XML-22006: Error while reading input XML URL (string).

XML-22007: Error while reading input XML reader (string).

XML-22008: Namespace prefix string used but not declared.

XML-22009: Attribute string not found in string.

XML-22010: Element string not found in string.

XML-22011: Cannot construct XML PI with content: string.

XML-22012: Cannot construct XML comment with content: string.

XML-22013: Error in expression: string.

XML-22014: Expecting node-set before relative location path.

XML-22015: Function string not found.

XML-22016: Extension function namespace should start with string.

XML-22017: Literal expected in string function. Found string.

XML-22018: Parse Error in string function.

XML-22019: Expected string instead of string.
XML-22020: Error in extension function arguments.


XML-22022: Error while testing predicates. Not a nodeset type.

XML-22023: Literal Mismatch.

XML-22024: Unknown multiply operator.

XML-22025: Expression error: Empty string.

XML-22026: Unknown expression at EOF: string.

XML-22027: Closing } not found in Attribute Value template.

XML-22028: Expression value type string not recognized by string.

XML-22029: Cannot transform child string in string.

XML-22030: Attribute value string not expected for string.

XML-22031: Variable not defined: string.

XML-22032: Found a single } outside expression in Attribute value template.

XML-22033: Token not recognized:!. 

XML-22034: Namespace definition not found for prefix string.

XML-22035: Axis string not found
XML-22036: Cannot convert string to string.


XML-22039: Extension function error: Error invoking constructor for string

XML-22040: Extension function error: Overloaded constructors for string

XML-22041: Extension function error: Constructor not found for string

XML-22042: Extension function error: Overloaded method string

XML-22043: Extension function error: Method not found string

XML-22044: Extension function error: Error invoking string:string

XML-22045: Extension function error: Class not found string

XML-22046: Apply import cannot be called when current template is null.

XML-22047: Invalid instantiation of string in string context.

XML-22048: The string element children must precede all other element children of an string element.

XML-22049: Template string invoked but not defined.

XML-22050: Duplicate variable string definition.
XML-22051: only a literal or a reference to a variable or parameter is allowed in id() function when used as a pattern

XML-22052: no sort key named as: string was defined

XML-22053: cannot detect encoding in unparsed-text(), please specify

XML-22054: no such xsl:function with namespace: string and local name: string was defined

XML-22055: range expression can only accept xs:integer data type, but not string

XML-22056: exactly one of four group attributes must be present in xsl:for-each-group

XML-22057: string can only have string as children

XML-22058: wrong child of xsl:function

XML-22059: wrong child order of xsl:function

XML-22060: TERMINATE PROCESSING

XML-22061: teminate attribute in <xsl:message> can only be yes or no

XML-22062: string must have at least one string child

XML-22063: no definition for character-map with qname string

XML-22064: cannot define character-map with the same name string and the same import precedence

Cause: A required child was not found.

Action: After error msgfreeze is over, throws an error (without the required child element, it can do nothing).
XML-22065: at least one *string* must be defined under *string*
Cause: a required child is missing.
Action: without the required child, it can do nothing, so throws an error.

XML-22066: if select attribute is present, *string* instructions sequence-constructor must be empty
Cause: the "select" attribute and sequence constructor should be mutually exclusive for this instruction.
Action: None. Throw an error.

XML-22067: if use attribute is present, *string* instructions sequence-constructor must be empty
Cause: the "use" attribute and sequence constructor should be mutually exclusive for this instruction.
Action: None. Throw an error.

XML-22068: only primary sort key is allowed to have the stable attribute.
Cause: the secondary sort key has a stable attribute.
Action: None. Throw an error.

XML-22069: only *string* or *string* is allowed.
Cause: user's typo.
Action: None. Throw an error.

XML-22101: DOMSource node as this type not supported.

XML-22103: DOMResult can not be this kind of node.

XML-22106: Invalid StreamSource - InputStream, Reader, and SystemId are null.

XML-22107: Invalid SAXSource - InputSource is null.

XML-22108: Invalid Source - URL format is incorrect.

XML-22109: Internal error while reporting SAX events.

XML Error Messages

XML-22111: Invalid Result set in TransformerHandler.

XML-22112: Namespace URI missing }.

XML-22113: Namespace URI should start with {.

XML-22117: URL format has problems (null or bad format or missing 'href' or missing '=').

XML-22121: Could not get associated stylesheet.

XML-22122: Invalid StreamResult - OutputStream, Writer, and SystemId are null.

XML-22900: An internal error condition occurred.

XPath Error Messages

These error messages are in the range XML-23000 through XML-23999.

XML-23002: internal xpath error
Cause: This was an error returned by the XPath/XQuery datamodel or XPathF&O.
Action: Check the XPath expression.

XML-23003: XPath 2.0 feature schema-element/schema-attribute not supported
Cause: This error was caused by using the kindtest schema-element or schema-attribute. These are not supported for this release.
Action: Remove usage of schema-element or schema-attribute kindtest

XML-23006: value does not match required type
Cause: During the evaluation phase, there was a type error as the value did not match a required type specified by the matching rules in XPath 2.0 SequenceType Matching.
Action: Modify the stylesheet to reflect the correct type.

XML-23007: FOAR0001: division by zero
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23008: FOAR0002: numeric operation overflow/unflow
Cause: This was an XPath 2.0 F&O specification error.
**Action**: Check the XPath expression.

**XML-23009: FOCA0001: Error in casting to decimal**  
**Cause**: This was an XPath 2.0 F&O specification error.  
**Action**: Check the XPath expression.

**XML-23010: FOCA0002: invalid lexical value**  
**Cause**: This was an XPath 2.0 F&O specification error.  
**Action**: Check the XPath expression.

**XML-23011: FOCA0003: input value too large for integer**  
**Cause**: This was an XPath 2.0 F&O specification error.  
**Action**: Check the XPath expression.

**XML-23012: FOCA0004: Error in casting to integer**  
**Cause**: This was an XPath 2.0 F&O specification error.  
**Action**: Check the XPath expression.

**XML-23013: FOCA0005: NaN supplied as float/double value**  
**Cause**: This was an XPath 2.0 F&O specification error.  
**Action**: Check the XPath expression.

**XML-23014: FOCH0001: invalid codepoint**  
**Cause**: This was an XPath 2.0 F&O specification error.  
**Action**: Check the XPath expression.

**XML-23015: FOCH0002: unsupported collation**  
**Cause**: This was an XPath 2.0 F&O specification error.  
**Action**: Check the XPath expression.

**XML-23016: FOCH0003: unsupported normalization form**  
**Cause**: This was an XPath 2.0 F&O specification error.  
**Action**: Check the XPath expression.

**XML-23017: FOCH0004: collation does not support collation units**  
**Cause**: This was an XPath 2.0 F&O specification error.  
**Action**: Check the XPath expression.

**XML-23018: FODC0001: no context document**  
**Cause**: This was an XPath 2.0 F&O specification error.  
**Action**: Check the XPath expression.

**XML-23019: FODC0002: Error retrieving resource**  
**Cause**: This was an XPath 2.0 F&O specification error.
XML Error Messages

Action: Check the XPath expression.

XML-23020: FODC0003: Error parsing contents of resource
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23021: FODC0004: invalid argument to fn:collection()
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23022: FODT0001: overflow in date/time arithmetic
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23023: FODT0002: overflow in duration arithmetic
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23024: FONC0001: undefined context item
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23025: FONS0002: default namespace is defined
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23026: FONS0003: no prefix defined for namespace
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23027: FONS0004: no namespace found for prefix
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23028: FONS0005: base URI not defined in the static context
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23029: FORG0001: invalid value for cast/constructor
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23030: FORG0002: invalid argument to fn:resolve-uri()
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23031: FORG0003: zero-or-one called with sequence containing more than one item
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23032: FORG0004: fn:one-or-more called with sequence containing no items
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23033: FORG0005: exactly-one called with sequence containing zero or more than one item
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23034: FORG0006: invalid argument type
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23035: FORG0007: invalid argument to aggregate function
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23036: FORG0008: both arguments to fn:dateTime have a specified timezone
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23037: FORG0009: base uri argument to fn:resolve-uri is not an absolute URI
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23038: FORX0001: invalid regular expression flags
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23039: FORX0002: invalid regular expression
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23040: FORX0003: regular expression matches zero-length string
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.
XML-23041: FORX0004: invalid replacement string
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23042: FOTY0001: type error
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23043: FOTY0011: context item is not a node
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23044: FOTY0012: items not comparable
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23045: FOTY0013: type does not have equality defined
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23046: FOTY0014: type exception
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23047: FORT0001: invalid number of parameters
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23048: FOTY0002: type definition not found
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23049: FOTY0021: invalid node type
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23050: FOER0000: unidentified error
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML-23051: FODC0005: invalid argument to fn:doc
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.
XML-23052: FODT0003: invalid timezone value
Cause: This was an XPath 2.0 F&O specification error.
Action: Check the XPath expression.

XML Schema Validation Error Messages
These error messages are in the range XML-24000 through XML-24099.

XML-24000: internal error
Cause: An unexpected error occurred during processing
Action: Report the error

XML-24001: attribute string not expected at line string, column string
Cause: [cvc-assess-attr.1] The attribute were not expected for owner element
Action: Add the attribute declaration to the type of the owner element

XML-24002: can not find element declaration string.
Cause: [cvc-assess-elt.1.1.1.1] The element declaration required by processor for validation was absent.
Action: Add the element declaration to schema, or change the instance document to comply to schema.

XML-24003: context-determined element declaration string absent.
Cause: [cvc-assess-elt.1.1.1.2] The element declaration required by context was missing in schema
Action: Add the element declaration to schema

XML-24004: declaration for element string absent.
Cause: [cvc-assess-elt.1.1.1.3] The context-determined declaration was not skip and the declaration that matches the element could not be found in schema
Action: Add the element declaration to schema or change the context-determined declaration to skip

XML-24005: element string not assessed
Cause: [cvc-assess-elt.2]

XML-24006: element string laxly assessed
Cause: [cvc-assess-elt.2]

XML-24007: missing attribute declaration string in element string
Cause: [cvc-attribute.1] Attribute declaration was absent from element declaration
Action: Add the attribute declaration to schema.

XML-24008: type absent for attribute string
Cause: [cvc-attribute.2] Missing type definition for the attribute declaration
Action: Specify a data type for the attribute declaration.
XML-24009: invalid attribute value *string*
*Cause:* [cvc-attribute.3] Invalid attribute value with respect to its type
*Action:* Correct the attribute value in instance.

XML-2410: attribute value *string* and fixed value *string* not match
*Cause:* [cvc-au] Attribute's normalized value was not the same as the fixed value declared.
*Action:* Change attribute value to the required value.

XML-2411: type of element *string* is abstract.
*Cause:* [cvc-complex-type.1] The type of this element was specified as abstract.
*Action:* Remove the abstract attribute from the type definition.

XML-2412: no children allowed for element *string* with empty content type
*Cause:* [cvc-complex-type.2.1] The content type was specified empty while the actual content was not.
*Action:* Make the content empty or modify the content type of this element.

XML-2413: element child *string* not allowed for simple content
*Cause:* [cvc-complex-type.2.2] Element was declared with simple content, but instance had element children.
*Action:* Use only character content for this element.

XML-2414: characters *string* not allowed for element-only content
*Cause:* [cvc-complex-type.2.3] Characters appeared in the content of element with element-only content.
*Action:* Use only element children for this element.

XML-2415: multiple ID attributes in element *string* at line *string*, column *string*
*Cause:* [cvc-complex-type.2.5] More than one attributes with type ID or its derivation matched attribute wildcard.
*Action:* Do not use more than one attributes with ID or ID derived type.

XML-2416: invalid string value *string* at line *string*, column *string*
*Cause:* [cvc-datatype-valid.1.2.2] Characters were not valid with respect to string type.
*Action:* Correct the value to satisfy the declared type.

XML-2417: invalid boolean value *string* at line *string*, column *string*
*Cause:* [cvc-datatype-valid.1.2.2] Characters were not valid with respect to boolean type.
*Action:* Correct the value to satisfy boolean type, valid values are "0", "1", "true", and "false".

XML-2418: invalid decimal value *string* at line *string*, column *string*
*Cause:* [cvc-datatype-valid.1.2.2] Characters could not be parsed into a decimal value.
**Action:** Correct the data value to satisfy decimal type.

**XML-24019:** invalid float value *string* at line *string*, column *string*

**Cause:** [cvc-datatype-valid.1.2.2] Characters could not be parsed into a float value.

**Action:** Correct the value to satisfy string type

**XML-24020:** invalid double value *string* at line *string*, column *string*

**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid double format as specified in IEEE 754-1985.

**Action:** Correct the value to satisfy double format.

**XML-24021:** invalid duration *string* at line *string*, column *string*

**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in correct extended date time format defined in ISO 8601.

**Action:** Correct the value to satisfy format PnYnMnDTnHnMnS.

**XML-24022:** invalid date value *string* at line *string*, column *string*

**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid calendar date format specified in ISO 8601.

**Action:** Correct the value to satisfy CCYY-MM-DD format.

**Comments:** cvc-datatype-valid.1.2.2

**XML-24023:** invalid dateTime value *string* at line *string*, column *string*

**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid combined data time format as specified in ISO 8601.

**Action:** Correct the value to satisfy format CCYY-MM-DDThh:mm:ss with optional timezone.

**XML-24024:** invalid time value *string* at line *string*, column *string*

**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid time format as specified in ISO 8601.

**Action:** Correct the value to satisfy foramt DDThh:mm:ss with optional timezone.

**XML-24025:** invalid gYearMonth value *string* at line *string*, column *string*

**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid right-truncated date format, as specified in ISO 8601.

**Action:** Correct the value to satisfy format CCYY-MM.

**XML-24026:** invalid gYear value *string* at line *string*, column *string*

**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid right-truncated date format, as specified in ISO 8601.

**Action:** Correct the value to satisfy format CCYY.

**XML-24027:** invalid gMonthDay value *string* at line *string*, column *string*

**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid left-truncated date format, as specified in ISO 8601.
XML Error Messages

**Action:** Correct the value to required format --MM-DD.

**XML-24028:** invalid gDay value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid left-truncated date format, as specified in ISO 8601.
**Action:** Correct the value to required format ---DD.

**XML-24029:** invalid gMonth value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid left-and-right-truncated date format, as specified in ISO 8601.
**Action:** Correct the value to required format --MM--.

**XML-24030:** invalid hexBinary value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid hex encoded binary.
**Action:** Correct the value to satisfy hexBinary type

**XML-24031:** invalid base64Binary value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid with respect to base64 encoding.
**Action:** Correct the value to satisfy base64 binary encoding.

**XML-24032:** invalid anyURI value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid format as specified in RFC 2396 and RFC 2732.
**Action:** Correct the value to satisfy anyURI type

**XML-24033:** invalid QName value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not in valid QName format.
**Action:** Correct the value to satisfy QName type

**XML-24034:** invalid NOTATION value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for NOTATION type.
**Action:** Correct the value to satisfy NOTATION type

**XML-24035:** invalid normalizedString value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid normalizedString value.
**Action:** Correct the value to satisfy normalizedString type

**XML-24036:** invalid token value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for token type.
**Action:** Correct the value to satisfy token type

**XML-24037:** invalid language value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for language type.
**XML Error Messages**

**Action:** Correct the value to satisfy language type

**XML-24038:** invalid NMTOKEN value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for NMTOKEN type.
**Action:** Correct the value to satisfy NMTOKEN type

**XML-24039:** invalid NMTOKENS value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid list of NMTOKEN type.
**Action:** Correct the value to satisfy NMTOKENS type.

**XML-24040:** invalid Name value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for Name type.
**Action:** Correct the value to satisfy Name type

**XML-24041:** invalid NCName value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for NCName type.
**Action:** Correct the value to satisfy NCName type

**XML-24042:** invalid ID value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for ID type.
**Action:** Correct the value to satisfy ID type

**XML-24043:** invalid IDREF value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for IDREF type.
**Action:** Correct the value to satisfy IDREF type

**XML-24044:** invalid ENTITY value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for ENTITY type
**Action:** Correct the value to satisfy ENTITY type

**XML-24045:** invalid ENTITIES value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid list of ENTITY value.
**Action:** Correct the value to satisfy ENTITIES type

**XML-24046:** invalid integer value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for integer type.
**Action:** Correct the value to satisfy integer type

**XML-24047:** invalid nonPositiveInteger value *string* at line *string*, column *string*
**Cause:** [cvc-datatype-valid.1.2.2] Characters were not valid value for nonPositiveInteger type.
**Action:** Correct the value to satisfy nonPositiveInteger type
XML-24048: invalid negativeInteger value string
Cause: [cvc-datatype-valid.1.2.2] Characters were not valid value for negativeInteger type.
Action: Correct the value to satisfy negativeInteger type

XML-24049: invalid long value string at line string, column string
Cause: [cvc-datatype-valid.1.2.2] Characters were not valid value for long type.
Action: Correct the value to satisfy long type

XML-24050: invalid int value string at line string, column string
Cause: [cvc-datatype-valid.1.2.2] Characters were not valid value for int type.
Action: Correct the value to satisfy int type

XML-24051: invalid short value string at line string, column string
Cause: [cvc-datatype-valid.1.2.2] Characters were not valid value for short type.
Action: Correct the value to satisfy short type

XML-24052: invalid byte value string at line string, column string
Cause: [cvc-datatype-valid.1.2.2] Characters were not valid value for byte type.
Action: Correct the value to satisfy byte type

XML-24053: invalid nonNegativeInteger value string at line string, column string
Cause: [cvc-datatype-valid.1.2.2] Characters were not valid value for nonNegativeInteger type.
Action: Correct the value to satisfy nonNegativeInteger type

XML-24054: invalid unsignedLong value string at line string, column string
Cause: [cvc-datatype-valid.1.2.2] Characters were not valid value for unsignedLong type.
Action: Correct the value to satisfy unsignedLong type

XML-24055: invalid unsignedInt value string at line string, column string
Cause: [cvc-datatype-valid.1.2.2] Characters were not valid value for unsignedInt.
Action: Correct the value to satisfy unsignedInt type

XML-24056: invalid unsignedShort value string at line string, column string
Cause: [cvc-datatype-valid.1.2.2] Characters were not valid value for unsignedShort type.
Action: Correct the value to satisfy unsignedShort type

XML-24057: invalid unsignedByte value string at line string, column string
Cause: [cvc-datatype-valid.1.2.2] Characters were not valid value for unsignedByte type.
Action: Correct the value to satisfy unsignedByte type
XML-24058: value string must be valid with respect to one member type  
Cause: [cvc-datatype-valid.1.2.3] Characters were invalid with respect to any member type of union. 
Action: Correct data value to satisfy at least one member type

XML-24059: element string not expected at line string, column string 
Cause: [cvc-elt.1]

XML-24060: element string abstract 
Cause: [cvc-elt.2] Element declared abstract was used in instance document. 
Action: Do not declare the element as abstract.

XML-24061: element string not nillable 
Cause: [cvc-elt.3.1] There was an attribute xsi:nil, which was not allowed because the element declaration was not nillable. 
Action: Remove xsi:nil attribute from the element

XML-24062: no character or element children allowed for nil content string 
Cause: [cvc-elt.3.2.1] Element was specified nil but had character or element children. 
Action: Remove any element content or remove nil attribute.

XML-24063: nil element does not satisfy fixed value constraint 
Cause: [cvc-elt.3.2.2] Element had an fixed value while the content in instance was empty. 
Action: Remove nil attribute from element.

XML-24064: xsi:type not a QName at line string, column string 
Cause: [cvc-elt.4.1] The value of xsi:type attribute was not a QName. 
Action: Change the value to a valid QName that references to a type.

XML-24065: xsi:type string not resolved to a type definition 
Cause: [cvc-elt.4.2] The referenced type specified by xsi:type was absent. 
Action: Correct the value of xsi:type so it points to a valide type definition.

XML-24066: local type string not validly derived from the type of element string 
Cause: [cvc-elt.4.3] The type referenced by xsi:type was not derived from original type. 
Action: Modify the reference type definition so it satisfy the constraint, or use another type that is derived from original type.

XML-24067: value string not in enumeration 
Cause: [cvc-enumeration-valid] The value was not one in the enumeration constraint. 
Action: Use valid value specified in enumeration.

XML-24068: invalid facet string for type string 
Cause: [cvc-facet-valid] The given data value violates the constraining facet.
XML Error Messages

Action: Correct the data value.

**XML-24069: too many fraction digits in value string at line string, column string**

*Cause:* [cvc-fractionDigits-valid] The given number violated the fractionDigits constraining facet.

*Action:* Use fewer fraction digits.

**XML-24070: missing ID definition for ID reference string at line string, column string**

*Cause:* [cvc-id.1] There is no ID binding in the ID/IDREF table for validation root

*Action:* Define the ID for the ID reference

**XML-24071: duplicate ID string at line string, column string**

*Cause:* [cvc-id.2] Same ID was defined more than once.

*Action:* Eliminate duplicate ID attributes.

**XML-24072: duplicate key sequence string**


*Action:* Correct the document to make key sequence unique, or modify xpath to avoid it.

**XML-24073: target node set not equals to qualified node set for key string**

*Cause:* [cvc-identity-constraint.4.2.1] There were empty key sequences in key constraint.

*Action:* Make sure every element in target node set has a non-empty key sequence.

**XML-24074: element member string in key sequence is nillable**

*Cause:* [cvc-identity-constraint.4.2.3] The element selected as a member in a key sequence was nillable, which is not allowed.

*Action:* Modify the schema to make corresponding element declaration not nillable.

**XML-24075: missing key sequence for key reference string**

*Cause:* [cvc-identity-constraint.4.3] A keyref referenced to empty key sequence.

*Action:* Make sure every key sequence for keyref is has a corresponding key sequence for referenced key.

**XML-24076: incorrect length of value string**

*Cause:* [cvc-length-valid] The length of the value was not the same as specified in length facet.

*Action:* Use data value with correct length.

**XML-24077: value string greater than or equal to maxExclusive**

*Cause:* [cvc-maxExclusive-valid] The data value was out of boundary specified in maxExclusive facet.

*Action:* Correct the data value.
XML-24078: value string greater than the maxInclusive
Cause: [cvc-maxInclusive-valid] The data value was out of boundary specified in maxInclusive facet.
Action: Correct the data value.

XML-24079: value length of string greater than maxLength
Cause: [cvc-maxLength-valid] The length of the data value was greater than maxLength.
Action: Make the data value's length smaller than maxLength.

XML-24080: value string smaller or equals to minExclusive
Cause: [cvc-minExclusive-valid] The data value was out of lower boundary of value range.
Action: Use data value that is greater to minExclusive.

XML-24081: value string smaller than minInclusive
Cause: [cvc-minInclusive-valid] The data value was too small.
Action: Use data value not smaller than the value of minInclusive.

XML-24082: value string shorter than minLength
Cause: [cvc-minLength-valid] The length of value was smaller than that specified in minLength.
Action: Use data value with length greater than or equals to minLength.

XML-24083: wildcard particle in the content of element string not done
Cause: [cvc-particle.1.1] The wildcard particle's minOccurs had not been met.
Action: Have more elements in the content that match the wildcard.

XML-24084: element particle string not done
Cause: [cvc-particle.1.2] The element particle's minOccurs had not been met.
Action: Have more elements that match the element declaration or members in its substitution group.

XML-24085: model group string in the content of element string not done
Cause: [cvc-particle.1.3] The model group particle's minOccurs had not been met.
Action: Have more elements in the content that match the model group.

XML-24086: invalid literal string with respect to pattern facet string
Cause: [cvc-pattern-valid] The literal did not match the pattern constraining facet.
Action: Correct the lexical data to match pattern facet.

XML-24087: undefined type string
Cause: [cvc-resolve-instance.1] Could not resolve the type reference to a type definition
Action: Add the type definition to schema
XML-24088: undeclared attribute string
Cause: [cvc-resolve-instance.2] Could not resolve attribute reference to an attribute declaration.
Action: Add the attribute declaration to schema.

XML-24089: undeclared element string
Cause: [cvc-resolve-instance.3] Could not resolve element reference to an element declaration.
Action: Add the element declaration to schema.

XML-24090: undefined attribute group string
Cause: [cvc-resolve-instance.4] Could not resolve the attribute group reference to an attribute group definition.
Action: Define the attribute group definition in schema.

XML-24091: undefined model group string
Cause: [cvc-resolve-instance.5] Could not resolve the model group reference to a model group definition.
Action: Define the model group in schema.

XML-24092: undeclared notation string
Cause: [cvc-resolve-instance.6] Could not resolve the notation reference to a notation declaration.
Action: Add the notation declaration to schema.

XML-24093: too many digits in value string at line string, column string
Cause: [cvc-totalDigits-valid] The number of digits in numeric value was greater than the value of totalDigits facet.
Action: Use smaller numbers.

Schema Representation Constraint Error Messages
These error messages are in the range XML-24100 through XML-24199.

XML-24100: element string must belong to XML Schema namespace
Cause: Element in XML Schema document did not have Schema namespace.
Action: Specify XML Schema namespace http://www.w3.org/2001/XMLSchema

XML-24101: can not build schema from location string
Cause: [schema_reference.2] Processor could not find schema from given schema location.
Action: Fix the schema location.

XML-24102: can not resolve schema by target namespace string
Cause: [schema_reference.3] Processor was unable to retrieve schema based on given namespace.
Action: Fix the schema namespace

**XML-24103: invalid annotation representation at line string, column string**

Cause: [src-annotation]

**XML-24104: multiple annotations at line string, column string**

Cause: [src-annotation] More than one annotation elements appeared in component.

Action: Remove extra annotation.

**XML-24105: annotation must be the first element at line string, column string**

Cause: [src-annotation] Annotation was not the first element in component.

Action: Move annotation to the begining of component content.

**XML-24106: attribute wildcard before attribute declaration at line string, column string**

Cause: The attribute wildcard appeared before attribute declarations.

Action: Move attribute wildcard to the end of declaration.

**XML-24107: multiple attribute wildcard**

Cause: [src-attribute.1] More than one anyAttributes were declared.

Action: Remove extra attribute wildcards.

**XML-24108: default string and fixed string both present**

Cause: [src-attribute.1] Both default and fixed attributes were present in attribute declaration.

Action: Remove either default or fixed attribute.

**XML-24109: default value string conflicts with attribute use string**

Cause: [src-attribute.2] Both default and use were present, and value for use is not optional.

Action: Remove either default or use value.

**XML-24110: missing name or ref attribute**

Cause: [src-attribute.3.1] Neither name nor ref attribute was present in declaration.

Action: Add name or ref to the declaration.

**XML-24111: both name and ref presented in attribute declaration**

Cause: [src-attribute.3.1] Name and ref attribute were both present in attribute declaration.

Action: Add name or ref to the declaration.

**XML-24112: ref conflicts with form, type, or simpleType child**

Cause: [src-attribute.3.2] The attribute was a reference, and form, type or simpleType child were specified.

Action: Either change ref to name, or remove form, type and/or childrens.
XML-24113: type attribute conflicts with simpleType child  
Cause: [src-attribute.4] Both type attribute and simpleType child were present.  
Action: Remove either type reference or type definition.

XML-24114: intersection of attribute wildcard is not expressible  
Cause: [src-attribute_group.2] Attributes wildcards defined were not expressible with a wildcard.  
Action: Remove inexpressible attribute wildcards.

XML-24115: circular attribute group reference string  
Cause: [src-attribute_group.3] Attribute group were circularly referenced outside redefine.  
Action: Remove circular reference.

XML-24116: circular group reference string  
Cause: group were circularly referenced outside redefine.  
Action: Remove circular reference.

XML-24117: base type string for complexContent is not complex type  
Cause: [src-ct.1] Derived a complexType with complex content from simple type.  
Action: Change base type to complex type.

XML-24118: simple content required in base type string  
Cause: [src-ct.2] A complexType with simpleContent was derived from a complexType with complex content.  
Action: Change base type to simple type (if derivation is extension) or simpleContent complex type.

XML-24119: properties specified with element reference string  
Cause: [src-element.2.2] Element reference also had complexType, simpleType, key, keyrefunique children or nillable, form, default, block, or type attribute.  
Action: Remove conflict attributes or children.

XML-24120: simpleType and complexType can not both present in element declaration string  
Cause: [src-element.3] Element declaration had both complexType, simpleType children.  
Action: Remove either simpleType or complexType child.

XML-24121: imported namespace string must different from namespace string  
Cause: [src-import.1.1] The namespace of import was the same as the target namespace of importing schema.  
Action: Change import to inclusion.
XML-24122: target namespace string required
Cause: [src-import.1.2] Imported namespace was specified but absent imported schema.
Action: Remove namespace attribute in element import, or add target namespaces to the imported schema.

XML-24123: namespace string is different from expected targetNamespace string
Cause: [src-import.3.1] Specified namespace was different from actual targetNamespace imported.
Action: Correct the namespace attribute in import element.

XML-24124: targetNamespace string not expected in schema
Cause: [src-import.3.2] Specified a no-namespace schema, but actual schema had targetNamespace.
Action: Remove the imported schema’s targetNamespace attribute.

XML-24125: can not include schema from string
Cause: [src-include.1] Processor was unable to include a schema from given location.
Action: Check correctness of URL and URL resolver.

XML-24126: included targetNamespace string must the same as string
Cause: [src-include.2.1] Tried to include a schema with different targetNamespace.
Action: Use import instead of include.

XML-24127: no-namespace schema can not include schema with target namespace string
Cause: [src-include.2.2] A schema without targetNamespace tried to include a schema with targetNamespace.
Action: Use import instead of include.

XML-24128: itemType attribute conflicts with simpleType child
Cause: [src-list-itemType-or-simpleType] Both itemType attribute and simpleType child were present in list simple type declaration.
Action: Remove either itemType attribute or simpleType child.

XML-24129: prefix of qname string can not be resolved
Cause: [src-qname] Prefix of a qname was present, but did not map to any in-scope namespace.
Action: Declare a namespace corresponding to the prefix.

XML-24130: redefined schema has different namespace. line string column string
Cause: Redefined schema’s targetNamespace was not the same as the targetNamespace of redefining schema.
Action: Correct the targetNamespace in redefined schema.
Comments: src-redefine.3.1
XML-24131: no-namespace schema can only redefine schema without targetNamespace
Cause: [src-redefine.3.2] A no-namespace schema tried to redefine a schema with namespace
Action: Remove the targetNamespace attribute from redefined schema.

XML-24132: type derivation string must be restriction
Cause: [src-redefine.5] A simpleType or complexType was present in redefine, but the derivation was not restriction.
Action: Change the type redefinition, make it a restriction.

XML-24132: type string must redefine itself at line string, column string
Cause: [src-redefine.5] A simpleType or complexType was present in redefine, but its base type was not itself.
Action: Change the base type to redefine itself.

XML-24133: group string can have only one self reference in redefinition
Cause: [src-redefine.6.1.1] A group was present in redefine and it had more than onereferences to itself in its content.
Action: Remove extra self references in the group redefinition.

XML-24134: self reference of group string must not have minOccurs or maxOccurs other than 1 in redefine
Cause: [src-redefine.6.1.2] A minOccurs or maxOccurs with value other than 1 was specified in a group self reference in redefine.
Action: Remove the minOccurs or maxOccurs attribute.

XML-24135: redefined group string is not a restriction of its orginal group
Cause: [src-redefine.6.2.2] A group presented in redefine, without self reference but was not a valid restriction of its original group.
Action: Modify the content of the group, make it a valid restriction of its original.

XML-24236: attribute group string can have only one self reference in redefinition
Cause: [src-redefine.7.1] An attributeGroup was present in redefine and it had more than oneself references in its content.
Action: Remove extra self references.

XML-24136: redefined attribute group string must be a restriction of its orginal group
Cause: [src-redefine.7.2.2] An attributeGroup presented in redefine, without self reference but was not a valid restriction of its original.
Action: Modify the content of the attribute group, make it valid restriction of its original.

XML-24137: restriction must not have both base and simpleType child
Cause:[src-restriction-base-or-simpleType]
**XML-24138: simple type restriction must have either base attribute or simpleType child**  
**Cause:** [src-simple-type.2] Both base and simpleType were absent in simple type restriction  
**Action:** Add either base attribute or simpleType child.

**XML-24139: neither itemType or simpleType child present for list**  
**Cause:** [src-simple-type.3] Missing itemType attribute or simpleType child in list definition.  
**Action:** Add either itemType or simpleType child.

**XML-24140: itemType and simpleType child can not both be present in list type.**  
**Cause:** [src-simple-type.3] Both itemType attribute and simpleType child were present in list definition.  
**Action:** Remove either itemType or simpleType child.

**XML-24141: circular union type is disallowed**  
**Cause:** [src-simple-type.4] Some member types in union type made references to the union type  
**Action:** Remove the circular references.

**XML-24142: facet string can not be specified more than once**  
**Cause:** [src-single-facet-value] Same facet other than enumeration and pattern had been specified more than once, which is not allowed.  
**Action:** Remove extra facets.

**XML-24143: memberTypes and simpleType child can not both be absent in union**  
**Cause:** [src-union-memberTypes-or-simpleTypes] Both memberTypes and simpleType were absent for a union type.  
**Action:** Either specify memberTypes or add simpleType children.

**XML-24144: facets can only used for restriction**  
**Cause:** [st-restrict-facets] Derivation was not restriction while facet children were present.  
**Action:** Remove facet children.

### Schema Component Constraint Error Messages

These error messages are in the range XML-24200 through XML-24399.

**XML-24201: duplicate attribute string declaration**  
**Cause:** [ag-props-correct.1] There were more than one attribute declarations with same namespace and name in attribute group definition.  
**Action:** Remove duplicate attribute declarations.
XML-24202: more than one attributes with ID type not allowed
Cause: [ag-props-correct.2] There were more than one attribute declarations with type ID.
Action: Change to other types for such attriubte declarations

XML-24203: invalid value constraint string
Cause: [a-props-correct.2] The fixed value or default value did not satisfy the attribute's type
Action: Use type valid default for fixed value.

XML-24204: value constraint string not allowed for ID type
Cause: [a-props-correct.3] Attribute with ID type had either fixed or default value constraint.
Action: Remove value constraint.

XML-24205: fixed value string does not match string in attribute declaration
Cause: [au-props-correct.2] Attribute reference specified a fixed value which is not the same as that in referenced declaration.
Action: Correct the fixed value to the same as specified in attribute declaration

XML-24206: value constraint must be fixed to match that in attribute declaration
Cause: [au-props-correct.2] Attribute reference specified a default value, while the referenced declaration had a fixed value.
Action: Remove default value form attribute reference.

XML-24207: invalid xpath expression string
Cause: [c-fields-xpaths.1] The value of xpath was not valid xpath expression as specified in XPath 1.0.
Action: Use correct xpath

XML-24208: invalid field xpath string
Cause: [c-fields-xpaths.2] The value of xpath did not satisfied field's restricted xpath syntax.
Action: Correct the xpath expression

XML-24209: maxOccurs in element string of All group must be 0 or 1
Cause: [cos-all-limited] Some elements in a All group had maxOccurs greater than one.

XML-24210: All group has to form a content type.
Cause: All group was contained in another model group
Action: Make all group at the top of a content type
Comments: cos-all-limited

XML-24211: All group has to form a content type.
Cause: [cos-applicable-facets] All group was contained in another model group
Action: Make all group at the top of a content type

**XML-24212: type string does not allow facet string**
Cause: [cos-applicable-facets] A facet not applicable to the simple type was used.
Action: Remove the facet.

**XML-24213: wildcard intersection is not expressible**
Cause: [cos-aw-intersect] Two wildcards in an attribute group had different negative namespaces
Action: Use only one wildcard with negative namespace

**XML-24214: base type not allow string derivation**
Cause: [cos-ct-derived-ok.1] Base type's final prevented the derivation.
Action: Remove the derivation method from the value of final in base type

**XML-24215: complex type string is not a derivation of type string**
Cause: [cos-ct-derived-ok.2] There was no derivation chain from base type to derived type.
Action: Fix the derivation chaining.

**XML-24216: must specify a particle in extened content type**
Cause: [cos-ct-extends.1.4.2.1] The content type of an extension of a complex type was empty
Action: Add particle to the content type of extension.

**XML-24217: content type string conflicts with base type's content type string**
Cause: [cos-ct-extends.1.4.2.2.2] Base type's content type was not empty and was not the same as the content type specified.
Action: Match the specified content type with that in base type.

**XML-24218: inconsistent local element declarations string**
Cause: More than one elements in the content had same name and namespace, but did not refer to same type.
Action: Make type references the same for all elements equal in name and namespace
Comments: cos-element-consistent

**XML-24219: element string is not valid substitutable for element string**
Cause: [cos-equiv-derived-ok-rec]

**XML-24220: itemType string can not be list**
Cause: [cos-list-of-atomic] The itemType of a list type was itself a list.
Action: Use atomic or union type as the itemType of list.
XML-24221: circular union string not allowed
Cause: [cos-no-circular-union] Union’s name and namespace matched one of its memberType.
Action: Remove any circular references

XML-24222: ambiguous particles string
Cause: [cos-nanambig] particles in a content type violated UPA (Unique Particle Attrition)constraint.
Action: Make content type particle unambiguous.

XML-24223: invalid particle extension
Cause: [cos-particle-extend]

XML-24224: invalid particle restriction
Cause: [cos-particle-restrict]

XML-24225: simple type string does not allowed restriction
Cause: [cos-st-derived-ok] Derivation was restriction but restriction was in base type’s final.
Action: Remove restriction from base type’s final.

XML-24226: invalid derivation from base type string
Cause: [cos-st-derived-ok] The derivation violated the "type derivation OK (simple)" constraint.
Action: Make the derivation satisfy the constraint.

XML-24227: atomic type can not restrict list string
Cause: [cos-st-restricts.1.1] base type is list,

XML-24228: base type can not be ur-type in restriction
Cause: [cos-st-restricts.1.1] Tried to directly restrict anySimpleType.

XML-24229: base type of list must be list or ur-type
Cause: [cos-st-restricts.2.3]

XML-24230: base type of union must be union or ur-type
Cause: [cos-st-restricts.3.3]

XML-24231: element default string requires mixed content to be emptiable
Cause: [cos-valide-default] Element had default constraint but its mixed content type was not emtiable.
Action: Remove default value constraint.

XML-24232: element default string requires mixed content or simple content
Cause: [cos-valide-default] Element had default value constraint but its content type was element only or empty.
**Action**: Remove default value constraint.

**XML-24233**: element default *string* must be valid to its content type  
**Cause**: [cos-valide-default] Element's default value constraint was invalid to its type.  
**Action**: Correct the default value or remove it.

**XML-24234**: wrong field cardinality for keyref *string*  
**Cause**: [c-props-correct] Number of fields were different between keyref and referenced key.  
**Action**: Ensure that keyref and referenced key have same number of fields.

**XML-24235**: complex type can only extend simple type *string*  
**Cause**: [ct-props-correct] Complex type was derived from simple type, but derivation was not extension.  
**Action**: Change restriction to extension.

**XML-24236**: cricular type definition *string*  
**Cause**: [ct-props-correct] Type was in its own derivation chain.  
**Action**: Remove recursive derivation.

**XML-24237**: base type *string* must be complex type  
**Cause**: [derivation-ok-restriction.1] Complex type was restricted from a simple type.  
**Action**: Change the restriction from a complex type.

**XML-24238**: attribute *string* not allowed in base type  
**Cause**: [derivation-ok-restriction.2] The attribute in restriction was not allowed for base type.  
**Action**: Correct the restriction of attribute use.

**XML-24239**: required attribute *string* not in restriction  
**Cause**: [derivation-ok-restriction.3] Requirement's attribute uses was not a subset of basetype's attribute uses.  
**Action**: Correct the restriction of attribute uses.

**XML-24240**: no correspoonding attribue wildcard in bas type *string*  
**Cause**: [derivation-ok-restriction.4] Restriction had an attribute wildcard that did not correspond to any attribute wildcard in base type.  
**Action**: Correct the derivation.

**XML-24241**: base type *string* must have simple content or emptiable  
**Cause**: [derivation-ok-restriction.5.1] Content type was simple, but the base type has complex content that is not mixed or not emptiable.  
**Action**: Change the content type from simple to element only.
XML-24242: base type string must have empty content or emptiable
Cause: [derivation-ok-restriction.5.2] Content type was empty, but the base type had simple content or not emptiable complex content.
Action: Change the content type from simple to element only.

XML-24243: enumeration facet required for NOTATION
Cause: [enumeration-required-notation] NOTATION type was used without enumeration facet.
Action: Specify enumeration facet for NOTATION.

XML-24244: invalid value string in enumeration
Cause: [enumeration-valid-restriction] Some value in enumeration was not valid in respect to the type.
Action: Correct invalid values.

XML-24245: default value string is element type invalid
Cause: [e-props-correct.2] Default value was invalid in respect to the type of element.
Action: Correct the default value.

XML-24246: invalid substitutionGroup string, type invalid
Cause: [e-props-correct.3] The type of the element was not a validly derivation from the type of element’s substitutionGroup.
Action: Correct the type or remove substitutionGroup.

XML-24247: ID type does not allow value constraint string
Cause: [e-props-correct.4] Type was ID or its derivation while there was a value constraint.
Action: Remove value constraint.

XML-24248: fractionDigits string greater than totalDigits string
Cause: [fractionDigits-totalDigits] The value for fractionDigits was greater than totalDigits.
Action: Make fractionDigits smaller or equal to totalDigits.

XML-24249: length facet can not be specified with minLength or maxLength
Cause: [length-minLength-maxLength] Both length and either minLength or maxLength were specified.
Action: Remove length facet.

XML-24250: length string not the same as length in base type’s
Cause: [length-valid-restriction] Specified a length that was not the same as the length in base type.
Action: Remove length facet.
XML-24251: maxExclusive greater than its original
Cause: [maxExclusive-valid-restriction] Restricted maxExclusive was greater than its original in base type.

XML-24252: minInclusive greater than or equal to maxExclusive
Cause: [maxInclusive-maxExclusive] Specified a minInclusive that was greater or equal to maxExclusive.
Action: Make minInclusive smaller than maxExclusive.

XML-24253: maxLength is greater than that in base type
Cause: [maxLength-valid-restriction] Specified a maxLength greater than original in base type.
Action: Specify a smaller maxLength to make it valid restriction.

XML-24254: circular group string disallowed
Cause: [mg-props-correct] Circular model group references.
Action: Remove circular references in model group definition.

XML-24256: minExclusive must be less than or equal to maxExclusive
Cause: [minExclusive-less-than-equals-to-maxExclusive] minExclusive was bigger than maxExclusive.
Action: Use smaller value for minExclusive.

XML-24257: minExclusive string must be less than maxInclusive
Cause: [minExclusive-less-than-maxInclusive] minInclusive specified was greater than or equal to maxInclusive.
Action: Specify smaller minInclusive.

XML-24258: invalid minExclusive string
Cause: [minExclusive-valid-restriction] Restriction's minExclusive was less than base type's minExclusive
Action: Specify greater value for minExclusive.

XML-24259: invalid minExclusive string
Cause: [minExclusive-valid-restriction] Restriction's minExclusive was less than base type's minInclusive
Action: Specify greater value for minExclusive.

XML-24260: invalid minExclusive string
Cause: [minExclusive-valid-restriction] Restriction's minExclusive was greater than base type's maxInclusive
Action: Specify smaller value for minExclusive.

XML-24261: invalid minExclusive string
Cause: [minExclusive-valid-restriction] Restriction's minExclusive was greater than or equals to base type's maxExclusive
Action: Specify smaller value for minExclusive.

**XML-24262: minInclusive string must not be greater than maxInclusive**

*Cause*: [minInclusive-less-than-equal-to-maxInclusive] Specified a minInclusive that was greater than maxInclusive
*Action*: Specify smaller value for minInclusive.

**XML-24263: Can not specify both minInclusive and minExclusive**

*Cause*: [minInclusive-minExclusive] Restriction specified both minInclusive and minExclusive.
*Action*: Remove either minInclusive or minExclusive.

**XML-24264: invalid minInclusive string**

*Cause*: [minInclusive-valid-restriction] Restriction's minInclusive was less than or equal to minInclusive in base type.
*Action*: Use minInclusive larger than that of base type.

**XML-24265: invalid minInclusive string**

*Cause*: [minInclusive-valid-restriction] Restriction's minInclusive was less than minExclusive in base type.
*Action*: Use minInclusive larger than or equal to the minExclusive of base type.

**XML-24266: invalid minInclusive string**

*Cause*: [minInclusive-valid-restriction] Restriction's minInclusive was greater than maxInclusive in base type.
*Action*: Use minInclusive smaller than or equal to the maxInclusive of base type.

**XML-24267: invalid minInclusive string**

*Cause*: Restriction's minInclusive was greater than or equal to maxInclusive in base type.
*Action*: Use minInclusive smaller than the maxInclusive of base type.
*Comments*: minInclusive-valid-restriction

**XML-24268: invalid minLength string**

*Cause*: [minLength-less-than-equal-to-maxLength] minLength in restriction is greater than base type's maxLength.
*Action*: Make minLength within the length range of base type.

**XML-24269: invalid minLength string**

*Cause*: [minLength-valid-restriction] Value of minLength is smaller than that of base type in restriction.
*Action*: Use bigger value for minLength.

**XML-24270: invalid minOccurs**

*Cause*: [no-xmlns] Declared an attribute with name xmlns.
Action: Remove such declaration.

**XML-24272: no xsi for targetNamespace**


*Action:* Use other target namespace.

**XML-24272: minOccurs is greater than maxOccurs**

*Cause:* [n-props-correct] The minOccurs of particle was greater than the maxOccurs.

*Action:* Use smaller value for minOccurs.

**XML-24281: maxOccurs must greater than or equal to 1**

*Cause:* [p-props-correct] The maxOccurs of particle was less than 1.

*Action:* Use greater value for maxOccurs.

**XML-24282: incorrect Notation properties**

*Cause:* [n-props-correct] The Notation declaration had incorrect properties.

*Action:* Fix Notation declaration.

**XML-24283: particle's range is not valid restriction**

*Cause:* [range-ok] Range of restriction was not within the range of parent particle.

**XML-24284: sequence group is not valid derivation of choice group**

*Cause:* Restriction did not satisfy constraint: Particle Derivation OK (Sequence:Choice -- MapAndSum)

*Comments:* rcase-MapAndSum

**XML-24285: element string is not valid restriction of element string**

*Cause:* [rcase-NameAndTypeOK] Restriction did not satisfy constraint: Particle Restriction OK

**XML-24286: element string is not valid restriction of wildcard**

*Cause:* [rcase-NSCompat] Restriction did not satisfy constraint: Particle Restriction OK

**XML-24287: group is not valid restriction of wildcard**

*Cause:* [rcase-NSRecurseCheckCardinality] Restriction did not satisfy constraint: Particle Restriction OK

**XML-24288: group any is not valid restriction**

*Cause:* [rcase-NSSubset] Restriction did not satisfy constraint: Particle Restriction OK(Any:Any -- NSSubset)

**XML-24289: invalid restriction of all or sequence group**

*Cause:* [rcase-Recurse] Restriction did not satisfy constraint: Particle Restriction OK(All:All, Sequence:Sequence: Recurse)
XML-24290: wildcard is not valid restriction
Cause: [rcase-RecurseLax] The wildcard was not validly restricted from another wildcard.

XML-24291: sequence is not a valid restriction of all
Cause: Restriction violated constraint: Particle Derivation OK (Sequence:All--RecurseUnordered)
Action: Fix the restriction.
Comments: rcase-RecurseUnordered

XML-24292: duplicate component definitions string
Cause: [sch-props-correct] There were two schema components with same name and namespace.
Action: Remove duplicate definitions.

XML-24293: Incorrect simple type definition properties
Cause: [st-props-correct]

XML-24294: wildcard is not a subset of its super
Cause: [w-props-correct] The namespace constraint was not a restriction of its super
Action: Correct namespace constraint.

XML-24295: totalDigits string is greater than string in base type
Cause: [totalDigits-valid-restriction] Restriction specified a totalDigits with value greater than that in base type.
Action: Use smaller value for totalDigits.

XML-24296: whiteSpace string can not restrict base type's string
Cause: [whiteSpace-valid-restriction] Restriction's whiteSpace was replace or preserve, and base had whiteSpace collapse, or restriction had replace while base had preserve.
Action: Eliminate conflict whiteSpace values.

XML-24297: circular substitution group string
Cause: Substitution group was circular.
Action: Remove the circular substitution group

XSQL Server Pages Error Messages
These error messages are in the range XML-25000 through XML-25999.

XML-25001: Cannot locate requested XSQL file. Check the name.

XML-25002: Cannot acquire database connection from pool: string
XML-25003: Failed to find config file *string* in CLASSPATH.

XML-25004: Could not acquire a database connection named: *string*

XML-25005: XSQL page is not well-formed.

XML-25006: XSLT stylesheet is not well-formed: *string*

XML-25007: Cannot acquire a database connection to process page.

XML-25008: Cannot find XSLT Stylesheet: *string*

XML-25009: Missing arguments on command line

XML-25010: Error creating: *string*
Using standard output.

XML-25011: Error processing XSLT stylesheet: *string*

XML-25012: Cannot Read XSQL Page

XML-25013: XSQL Page URI is null; check exact case of file name.

XML-25014: Resulting page is an empty document or had multiple document elements.

XML-25015: Error inserting XML Document

XML-25016: Error parsing posted XML Document

XML-25017: Unexpected Error Occurred
XML-25018: Unexpected Error Occurred processing stylesheet *string*

XML-25019: Unexpected Error Occurred reading stylesheet *string*

XML-25020: Config file *string* is not well-formed.

XML-25021: Serializer *string* is not defined in XSQL configuration file

XML-25022: Cannot load serializer class *string*

XML-25023: Class *string* is not an XSQL Serializer

XML-25024: Attempted to get response Writer after getting OutputStream

XML-25025: Attempted to get response OutputStream after getting Writer

XML-25026: Stylesheet URL references an untrusted server.

XML-25027: Failed to load *string* class for built-in xsql:*string* action.

XML-25028: Error reading *string*. Check case of the name.

XML-25029: Cannot load error handler class *string*

XML-25030: Class *string* is not an XSQL ErrorHandler

XML-25100: You must supply a *string* attribute.

XML-25101: Fatal error in Stylesheet Pool

XML-25102: Error instantiating class *string*
XML-25103: Unable to load class string

XML-25104: Class string is not an XSQLActionHandler

XML-25105: XML returned from PLSQL agent was not well-formed

XML-25106: Invalid URL string

XML-25107: Error loading URL string

XML-25108: XML Document string is not well-formed

XML-25109: XML Document returned from database is not well-formed

XML-25110: XML Document in parameter string is not well-formed

XML-25111: Problem including string

XML-25112: Error reading parameter value

XML-25113: Error loading XSL transform string

XML-25114: Parameter string has a null value

XML-25115: No posted document to process

XML-25116: No query statement supplied

XML-25117: No PL/SQL function name supplied

XML-25118: Stylesheet URL references an untrusted server.
XML Error Messages

XML-25119: You must supply either the string or string attribute.

XML-25120: You selected fewer than the expected string values.

XML-25121: Cannot use 'xpath' to set multiple parameters.

XML-25122: Query must be supplied to set multiple parameters

XML-25123: Error reading string. Check case of the name.

XML-25124: Error printing additional error information.

XML-25125: Only one of (string) attributes is allowed.

XML-25126: One of (string) attributes must be supplied.

XML Pipeline Error Messages

These error messages are in the range XML-30000 through XML-30999.

XML-30000: Error ignored in string: string
Cause: Error occurred while processes execution is ignored
Action: None required

XML-30001: Error occurred in execution of Process
Cause: Component being wrapped by pipeline process is causing error
Action: Might need to fix input xml content

XML-30002: Only XML type(s) string allowed.
Comments: Should not occur normally

XML-30003: Error creating/writing to output string
Cause: Output url provided might be invalid

XML-30004: Error creating base url string
Cause: URL provided as base url is invalid
Action: Fix base url provided
XML-30005: Error reading input *string*
Cause: Input url provided might be invalid

XML-30006: Error in processing pipedoc Error element

XML-30007: Error converting output to xml type required by dependent process

XML-30008: A valid parameter target is required
Cause: Param with name target is missing or invalid
Action: Please add param target pointing to the target output label

XML-30009: Error piping output to input

XML-30010: Process definition element *string* needs to be defined
Cause: Element procdef is missing
Action: Please add process definition to pipedoc

XML-30011: ContentHandler not available
Cause: The dependent process does not provide a valid ContentHandler
Action: Please implement the getContentHandler API in your Process.

XML-30012: Pipeline components are not compatible
Cause: Component output and input don't match in terms of document/docfrag
Action: Fix the pipedoc to use components which are compatible

XML-30013: Process with output label *string* not found
Cause: Process whose output label matched target label is not available
Action: Create a process in the pipedoc, where the output label matches the label of the target param

XML-30014: Pipeline is not complete, missing output/outparam label called *string*
Cause: A dependent process output label has not been named correctly, or a dependent process is missing
Action: Please make sure every dependent input has a corresponding output

XML-30016: Unable to instantiate class
Cause: A process could not be create as there is an error in the process definition element associated with it
Action: Correctly specify the class for a process definition
XML-30017: Target is up-to-date, pipeline not executed
Cause: Either the target does not exist, or the pipeline inputs are more recent than the target
Action: Use the ‘force’ option to execute pipeline regardless of whether the target is up-to-date

Java API for XML Binding (JAXB) Error Messages
These error messages are in the range XML-32000 through XML32999.

XML-32202: a problem was encountered because multiple <schemaBindings> were defined.
Cause: There was more than one instance of <schemaBindings> declaration in the annotation element of the <schema> element.
Action: Update the annotation to remove duplicate <schemaBinding> declaration.

XML-32203: a problem was encountered because multiple <class> name annotations were defined on node string.
Cause: There was more than one instance of <class> declaration in the annotation element of the node.
Action: Update the annotation to remove duplicate <class> declaration.

XML-32204: a problem was encountered because the name in <class> declaration contained a package name prefix string which was not allowed.
Cause: A failure occurred because the name attribute in the <class> declaration contained a package prefix.
Action: Update the className in <class> declaration.
Comments: The package prefix is inherited from the current value of package.

XML-32205: a problem was encountered because the property customization was not specified correctly on node string.
Cause: A failure occurred because the property customization was not specified correctly.
Action: Update the <property> customization.

XML-32206: a problem was encountered because the javaType customization was not specified correctly on node string.
Cause: A failure occurred because the property customization was not specified correctly.
Action: Update the <javaType> customization.

XML-32207: a problem was encountered in declaring the baseType customization on the node string.
Cause: A failure occurred because the baseType customization was not specified correctly.
Action: Update the <baseType> customization.
XML-32208: a problem was encountered because multiple baseType customizations were declared on the node string.
Cause: A failure occurred because multiple "baseType" customizations were declared.
Action: Remove one of the <baseType> customization declaration.

XML-32209: a problem was encountered because multiple javaType customizations were declared on the node string.
Cause: A failure occurred because multiple "javaType" customizations were declared.
Action: Remove one of the <javaType> customization declaration.

XML-32210: a problem was encountered because invalid value was specified on customization of string.
Cause: A failure occurred because an invalid value was specified on the globalBindings customization declaration.
Action: Check and correct the globalBindings customization value.

XML-32211: a problem was encountered because incorrect <schemaBindings> customization was specified.
Cause: A failure occurred because an invalid value was specified on the schemaBindings customization.
Action: Check and correct the schemaBindings customization value.

XML-32212: the <class> customization did not support specifying the implementation class using implClass declaration. The implClass declaration specified on node string was ignored.
Cause: A warning occurred because the implClass customization declaration was not supported.

XML-32213: the <globalBindings> customization did not support specifying user specific class that implements java.util.List. The collectionType declaration was ignored.
Cause: A warning occurred because the user specific implementation class for java.util.List was not supported.

TXU Error Messages
These error messages may occur during the execution of TXU interfaces.

General TXU Error Messages
These error messages are in the range TXU-0001 through TXU-0099.

TXU-0001: Fatal Error

TXU-0002: Error
TXU-0003: Warning

DLF Error Messages

These error messages are in the range TXU-0100 through TXU-0199.

TXU-0100: parameter string in query string not found
Cause: There is not a placeholder for the parameter in the query
Action: Supply a parameter whose id can be found as an associated placeholder in the associated query

TXU-0101: incompatible attributes col and constant coexist at string in query string
Cause: Attributes 'col' and 'constant' cannot coexist
Action: Remove either 'col' or 'constant' attribute

TXU-0102: node string not found
Cause: The document lacks an expected node
Action: Supply the missing node

TXU-0103: element string lacks content
Cause: The element has no data
Action: Supply content

TXU-0104: element string with SQL string lacks col or constant attribute
Cause: The element lacks a required attribute of 'col' or 'constant'
Action: Supply either 'col' or 'constant' attribute

TXU-0105: SQL exception string while processing SQL string
Cause: An error occurred during the SQL execution
Action: Resolve the error in the SQL statement

TXU-0106: no data for column string selected by SQL string
Cause: The SQL query returned no data
Action: Supply data or modify your query

TXU-0107: datatype string not supported
Cause: An attempt to process an unsupported datatype was made
Action: Change the datatype to a supported one

TXU-0108: missing maxsize attribute for column string
Cause: The size-unit attribute is specified but maxsize is not.
Action: Supply the maxsize attribute, too
TXU-0109: a text length of string for string exceeds the allowed maximum of string
Cause: The length of the text data is too long
Action: Shorten the data so it fits in the limit, or enlarge the maxsize attribute and ensure the database column is large enough.

TXU-0110: undeclared column string in row string
Cause: A column in the data section is not declared in the columns section
Action: Modify the column name to a declared one.

TXU-0111: lacking column data for string in row string
Cause: A column is declared but the data is missing.
Action: Supply the col element whose name attribute matches the column name.

TXU-0112: undeclared query parameter string for column string
Cause: The query parameter refers to an undeclared column
Action: Specify a declared column.

TXU-0113: incompatible attribute string with a query on column string
Cause: A column with a query cannot have the specified attribute
Action: Remove either the attribute or query.

TXU-0114: DLF parse error (string) on line string, character string in string
Cause: The format is in error as reported
Action: Correct the erroneous part.

TXU-0115: The specified date string string has an invalid format
Cause: The specified date string does not match the specified formatstring.
Action: Make sure the date string is in an appropriate date format.

TransX Informational Messages
These error messages are in the range TXU-0200 through TXU-0299.

TXU-0200: duplicate row at string
Cause: A duplicate row exists in the database
Action: This message appears on the DuplicateRowException to inform applications of existence of one or more duplicate rows already stored in the database.

TransX Error Messages
These error messages are in the range TXU-0300 through TXU-0399.

TXU-0300: document string not found
Cause: The document could not be located
Action: Modify the document location or supply the document at the location.
TXU-0301: file string could not be read
Cause: An I/O error happened when reading the file
Action: Resolve the I/O problem

TXU-0302: archive string not found
Cause: The archive file could not be located
Action: Ensure that the CLASSPATH includes TransX correctly and only once

TXU-0303: schema string not found in string
Cause: The schema definition of DLF could not be located
Action: Obtain an unbroken copy of a TransX archive

TXU-0304: archive path for string not found
Cause: The path for the archive could not be determined
Action: Ensure that the CLASSPATH includes TransX correctly and only once

TXU-0305: no database connection on string call for string
Cause: The operation was attempted without a database connection
Action: Open a connection first

TXU-0306: null tablename given; access denied
Cause: The table name is not provided
Action: Specify a table name

TXU-0307: lookup-keys could not be determined string
Cause: The data dictionary is corrupted
Action: Restore the data dictionary

TXU-0308: binary file string not found
Cause: The file name is invalid
Action: Supply a good file name

TXU-0309: a file size of string exceeds the allowed maximum of 2,000 bytes
Cause: The file is too large
Action: Reduce the file size

Assertion Messages
These error messages are in the range TXU-0400 through TXU-0499.

TXU-0400: missing SQL statement element on string
Cause: An internal assertion was not successful
Action: Contact Oracle customer support
TXU-0401: missing node string
Cause: An internal assertion was not successful
Action: Contact Oracle customer support

TXU-0402: invalid flag string
Cause: An internal assertion was not successful
Action: Contact Oracle customer support

TXU-0403: internal error string
Cause: An internal assertion was not successful
Action: Contact Oracle customer support

TXU-0404: unexpected Exception string
Cause: An internal assertion was not successful
Action: Contact Oracle customer support

Usage Description Messages
These error messages are in the range TXU-0500 through TXU-0599.

TXU-0500: XML data transfer utility

TXU-0501: Parameters are as follows:

TXU-0502: JDBC connect string

TXU-0503: You can omit the connect string information through the '@' symbol.

TXU-0504: Then jdbc:oracle:thin:@ will be supplied.

TXU-0505: database username

TXU-0506: database password

TXU-0507: data file name or URL

TXU-0508: Options:
TXU-0509: update existing rows

TXU-0510: raise exception if a row is already existing

TXU-0511: print data in the predefined format

TXU-0512: save data in the predefined format

TXU-0513: print the XML to load

TXU-0514: print the tree for update

TXU-0515: omit validation

TXU-0516: validate the data format and exit without loading

TXU-0517: preserve whitespace

TXU-0518: Examples:

**XSU Error Messages**

These error messages may occur during the execution of the XSU interfaces.

**Keywords and Syntax for Error Messages**

These error messages are in the range XSUK-0001 through XSUK-0099.

- XSUK-0001: DOCUMENT
- XSUK-0002: ROWSET
- XSUK-0003: ROW
XSUK-0004: ERROR

XSUK-0005: num

XSUK-0006: item

Generic Error Messages
These error messages are in the range XSUE-0000 through XSUE-0099.

XSUE-0000: Internal Error -- Exception Caught string

XSUE-0001: Internal Error -- string

XSUE-0002: string is not a scalar column. The row id attribute can only get values from scalar columns.

XSUE-0003: string is not a valid column name.

XSUE-0004: This object has been closed. If you would like the object not to be closed implicitly between calls, see the string method.

XSUE-0005: The row-set enclosing tag and the row enclosing tag are both omitted; consequently, the result can consist of at most one row which contains exactly one column which is not marked to be an XML attribute.

XSUE-0006: The row enclosing tag or the row-set enclosing tag is ommitted; consequently to get a well formed XML document, the result can only consist of a single row with multiple columns or multiple rows with exactly one column which is not marked to be an XML attribute.

XSUE-0007: Parsing of the sqlname failed -- invalid arguments.

XSUE-0008: Character string is not allowed in an XML tag name.
XSU Error Messages

**XSUE-0009:** this method is not supported by *string* class. Please use *string* instead.

**XSUE-0010:** The number of bind names does not equal the number of bind values.

**XSUE-0011:** The number of bind values does not match the number of binds in the SQL statement.

**XSUE-0012:** Bind name identifier *string* does not exist in the sql query.

**XSUE-0013:** The bind identifier has to be of non-zero length.

**XSUE-0014:** Root node supplied is null.

**XSUE-0015:** Invalid LOB locator specified.

**XSUE-0016:** File *string* does not exit.

**XSUE-0017:** Can not create oracle.sql.STRUCT object of a type other than oracle.sql.STRUCT (i.e. ADT).

**XSUE-0018:** Null is not a valid DocumentHandler.

**XSUE-0019:** Null and empty string are not valid namespace aliases.

**XSUE-0020:** to use this method you will have to override it in your subclass.

**XSUE-0021:** You are using an old version of the gss library; thus, sql-xml name escaping is not supported.

**XSUE-0022:** cannot create XMLType object from opaque base type: *string*

---

**Query Error Messages**

These error messages are in the range XSUE-0100 through XSUE-0199.
XSUE-0100: Invalid context handle specified.

XSUE-0101: In the FIRST row of the resultset there is a nested cursor whose parent cursor is empty; when this condition occurs we are unable to generate a dtd.

XSUE-0102: string is not a valid IANA encoding.

XSUE-0103: The resultset is a "TYPE_FORWARD_ONLY" (non-scrollable); consequently, xsu can not reposition the read point. Furthermore, since the result set has been passed to the xsu by the caller, the xsu can not recreate the resultset.

XSUE-0104: input character is invalid for well-formed XML: string

DML Error Messages

These error messages are in the range XSUE-0200 through XSUE-0299.

XSUE-0200: The XML element tag string does not match the name of any of the columns/attributes of the target database object.

XSUE-0201: NULL is an invalid column name.

XSUE-0202: Column string, specified to be a key column, does not not exits in table string.

XSUE-0203: Column string, specified as column to be updated, does not exist in the table string.

XSUE-0204: Invalid REF element - string - attribute string missing.

XSUE-0206: Must specify key values before calling update routine. Use the string function.

XSUE-0207: UpdateXML: No columns to update. The XML document must contain some non-key columns to update.
XSU Error Messages

XSUE-0208: The key column array must be non empty.

XSUE-0209: The key column array must be non empty.

XSUE-0210: No rows to modify -- the row enclosing tag missing. Specify the correct row enclosing tag.

XSUE-0211: string encountered during processing ROW element string in the XML document.

XSUE-0212: string XML rows were successfully processed.

XSUE-0213: All prior XML row changes were rolled back.

Pieces of Error Messages

These error messages are in the range XSUE-0300 through XSUE-0400.

XSUE-0300: Note

XSUE-0301: Exception string caught: string

XSUE-0302: column

XSUE-0303: name

XSUE-0303: invalid

XSUE-0304: xml document

XSUE-0305: template
access control entry (ACE)
An entry in the access control list that grants or denies access to a given principal.

access control list (ACL)
A list of access control entries that determines which principals have access to a given resource or resources.

ACE
Access Control Entry. See access control entry.

ACL
Access Control List. See access control list.

application server
A server designed to host applications and their environments, permitting server applications to run. A typical example is Oracle Application Server, which is able to host Java, C, C++, and PL/SQL applications in cases where a remote client controls the interface.

attribute
A property of an element that consists of a name and a value separated by an equals sign and contained within the start-tags after the element name. In this example, `<Price units='USD'>5</Price>`, units is the attribute and USD is its value, which must be in single or double quotes. Attributes may reside in the document or DTD. Elements may have many attributes but their retrieval order is not defined.

Binary XML
An XML representation using the compact schema-aware format.

BLOB
See binary large object.

Business-to-Business (B2B)
A term describing the communication between businesses in the selling of goods and services to each other. The software infrastructure to enable this is referred to as an exchange.

Business-to-Consumer (B2C)
A term describing the communication between businesses and consumers in the selling of goods and services.
**callback**
A programmatic technique in which one process starts another and then continues. The second process then calls the first as a result of an action, value, or other event. This technique is used in most programs that have a user interface to allow continuous interaction.

**cartridge**
A stored program in Java or PL/SQL that adds the necessary functionality for the database to understand and manipulate a new datatype. Cartridges interface through the Extensibility Framework within Oracle Database version 8 or later. Oracle Text is such a cartridge, adding support for reading, writing, and searching text documents stored within the database.

**Cascading Style Sheets**
A simple mechanism for adding style (fonts, colors, spacing, and so on) to Web documents.

**CDATA**
See character data.

**character data (CDATA)**
Text in a document that should not be parsed is put within a **CDATA** section. This allows for the inclusion of characters that would otherwise have special functions, such as &, <, >, and so on. **CDATA** sections can be used in the content of an element or in attributes.

**child element**
An element that is wholly contained within another, which is referred to as its parent element. For example `<Parent><Child></Child></Parent>` illustrates a child element nested within its parent element.

**Class Generator**
A utility that accepts an input file and creates a set of output classes that have corresponding functionality. In the case of the XML class generator, the input file is a DTD and the output is a series of classes that can be used to create XML documents conforming with the DTD.

**CLASSPATH**
The operating system environmental variable that the JVM uses to find the classes it needs to run applications.

**Common Object Request Broker API (CORBA)**
An Object Management Group standard for communicating between distributed objects across a network. These self-contained software modules can be used by applications running on different platforms or operating systems. CORBA objects and their data formats and functions are defined in the Interface Definition Language (IDL), which can be compiled in a variety of languages including Java, C, C++, Smalltalk and COBOL.

**Common Oracle Runtime Environment (CORE)**
The library of functions written in C that provides developers the ability to create code that can be easily ported to virtually any platform and operating system.
CORBA
See Common Object Request Broker API.

CSS
See Cascading Style Sheets.

Database Access Descriptor (DAD)
A DAD is a named set of configuration values used for database access. A DAD specifies information such as the database name or the Oracle Net service name, the ORACLE_HOME directory, and Globalization Support configuration information such as language, sort type, and date language.

datagram
A text fragment, which may be in XML format, that is returned to the requester embedded in an HTML page from a SQL query processed by the XSQL Servlet.

DBUriType
The datatype used for storing instances of the datatype that permits XPath-based navigation of database schemas.

DOCTYPE
The term used as the tag name designating the DTD or its reference within an XML document. For example, <![DOCTYPE person SYSTEM "person.dtd"> declares the root element name as person and an external DTD as person.dtd in the file system. Internal DTDs are declared within the DOCTYPE declaration.

Document Location Hint
Oracle XML DB uses the Document Location Hint to determine which XML schemas are relevant to processing the instance document. It assumes that the Document Location Hint will map directly to the URL used when registering the XML schema with the database. When the XML schema includes elements defined in multiple namespaces, an entry must occur in the schemaLocation attribute for each of the XML schemas. Each entry consists of the namespace declaration and the Document Location Hint. The entries are separated from each other by one or more whitespace characters. If the primary XML schema does not declare a target namespace, then the instance document also needs to include a noNamespaceSchemaLocation attribute that provides the Document Location Hint for the primary XML schema.

Document Object Model (DOM)
An in-memory tree-based object representation of an XML document that enables programmatic access to its elements and attributes. The DOM object and its interface is a W3C recommendation. It specifies the Document Object Model of an XML Document including the APIs for programmatic access. DOM views the parsed document as a tree of objects.

Document Type Definition (DTD)
A set of rules that define the legal structure of an XML document. DTDs are text files that derive their format from SGML and can either be included in an XML document by using the DOCTYPE element or by using an external file through a DOCTYPE reference.

DOM
See Document Object Model.
DOM fidelity

To assure the integrity and accuracy of this data, for example, when regenerating XML documents stored in Oracle XML DB, Oracle XML DB uses a data integrity mechanism, called DOM fidelity. DOM fidelity refers to when the returned XML documents are identical to the original XML document, particularly for purposes of DOM traversals. Oracle XML DB assures DOM fidelity by using a binary attribute, SYS_XDBPD$.

DTD

See Document Type Definition.

EDI

Electronic Data Interchange.

element

The basic logical unit of an XML document that can serve as a container for other elements such as children, data, and attributes and their values. Elements are identified by start-tags, such as <name>, and end-tags, such as </name>, or in the case of empty elements, <name/>.

empty element

An element without text content or child elements. It can only contain attributes and their values. Empty elements are of the form <name/> or <name></name>, where there is no space between the tags.

Enterprise JavaBean (EJB)

An independent program module that runs within a JVM on the server. CORBA provides the infrastructure for EJBs, and a container layer provides security, transaction support, and other common functions on any supported server.

entity

A string of characters that may represent either another string of characters or special characters that are not part of the document character set. Entities and the text that is substituted for them by the parser are declared in the DTD.

eXtensible Markup Language (XML)

An open standard for describing data developed by the World Wide Web Consortium (W3C) using a subset of the SGML syntax and designed for Internet use.

eXtensible Stylesheet Language (XSL)

The language used within stylesheets to transform or render XML documents. There are two W3C recommendations covering XSL stylesheets—XSL Transformations (XSLT) and XSL Formatting Objects (XSLFO).

XSL consists of two W3C recommendations: XSL Transformations for transforming one XML document into another and XSL Formatting Objects for specifying the presentation of an XML document. XSL is a language for expressing stylesheets. It consists of two parts:

- A language for transforming XML documents (XSLT), and
- An XML vocabulary for specifying formatting semantics (XSLFO).

An XSL stylesheet specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary.
**eXtensible Stylesheet Language Formatting Object (XSLFO)**

The W3C standard specification that defines an XML vocabulary for specifying formatting semantics. See FOP.

**eXtensible Stylesheet Language Transformation (XSLT)**

Also written as XSL-T. The XSL W3C standard specification that defines a transformation language to convert one XML document into another.

**FOP**

Print formatter driven by XSL formatting objects. It is a Java application that reads a formatting object tree and then renders the resulting pages to a specified output. Output formats currently supported are PDF, PCL, PS, SVG, XML (area tree representation), Print, AWT, MIF and TXT. The primary output target is PDF.

**HASPATH**

The SQL operator that is part of Oracle Text and used for querying XMLType datatypes for the existence of a specific XPath.

**hierarchical indexing**

The data relating a folder to its children is managed by the Oracle XML DB hierarchical index, which provides a fast mechanism for evaluating path names similar to the directory mechanisms used by operating system filesystems. Any path name-based access will normally use the Oracle XML DB hierarchical index.

**HTTP**

See Hypertext Transport Protocol.

**HTTPS**

See Hypertext Transport Protocol, Secure.

**HTTPUriType**

The datatype used for storing instances of the datatype that permits XPath-based navigation of database schemas in remote databases.

**IDE**


**infoset**

XML Information Set, an abstract data set consisting of a number of information items. It has at least one information item: the document node, but the infoset is not necessarily valid XML. The W3C Recommendation is at http://www.w3.org/TR/xml-infoset/

**INPATH**

The SQL operator that is part of Oracle Text and is used for querying XMLType datatypes for searching for specific text within a specific XPath.

**instance document**

An XML document validated against an XML schema. If the instance document conforms to the rules of the schema, then it is said to be valid.
**instantiate**
A term used in object-based languages such as Java and C++ to refer to the creation of an object of a specific class.

**Integrated Development Environment (IDE)**
A set of programs designed to aid in the development of software run from a single user interface. JDeveloper is an IDE for Java development, because it includes an editor, compiler, debugger, syntax checker, help system, and so on, to permit Java software development through a single user interface.

**Multimedia**
The collection of complex datatypes and their access in Oracle. These include text, video, time-series, and spatial data.

**Internet Inter-ORB Protocol (IIOP)**
The protocol used by CORBA to exchange messages on a TCP/IP network such as the Internet.

**J2EE**
See Java 2 Platform, Enterprise Edition.

**Java**
A high-level programming language developed and maintained by Sun Microsystems where applications run in a virtual machine known as a JVM. The JVM is responsible for all interfaces to the operating system. This architecture permits developers to create Java applications that can run on any operating system or platform that has a JVM.

**Java 2 Platform, Enterprise Edition (J2EE)**
The Java platform (Sun Microsystems) that defines multitier enterprise computing.

**Java API for XML Processing (JAXP)**
Enables applications to parse and transform XML documents using an API that is independent of a particular XML processor implementation.

**Java Architecture for XML Binding (JAXB)**
API and tools that map to and from XML documents and Java objects. A JSR-31 recommendation.

**JavaBeans**
An independent program module that runs within a JVM, typically for creating user interfaces on the client. Also known as Java Bean. The server equivalent is called an Enterprise JavaBean (EJB). See also Enterprise JavaBean.

**Java Database Connectivity (JDBC)**
The programming API that enables Java applications to access a database through the SQL language. JDBC drivers are written in Java for platform independence but are specific to each database.

**Java Developer's Kit (JDK)**
The collection of Java classes, runtime, compiler, debugger, and usually source code for a version of Java that makes up a Java development environment. JDKs are designated by versions, and Java 2 is used to designate versions from 1.2 onward.
**Java Naming and Directory Interface (JNDI)**
A programming interface from Sun for connecting Java programs to naming and directory services such as DNS, LDAP, and NDS. Oracle XML DB Resource API for Java/JNDI supports JNDI.

**Java Runtime Environment (JRE)**
The collection of compiled classes that make up the Java virtual machine on a platform. JREs are designated by versions, and Java 2 is used to designate versions from 1.2 onward.

**JavaServer Pages (JSP)**
An extension to the servlet functionality that enables a simple programmatic interface to Web pages. JSPs are HTML pages with special tags and embedded Java code that is executed on the Web server or application server providing dynamic functionality to HTML pages. JSPs are actually compiled into servlets when first requested and run in the JVM of the server.

**Java Specification Request (JSR)**
A recommendation of the Java Community Process organization (JCP), such as JAXB.

**Java Virtual Machine (JVM)**
The Java interpreter that converts the compiled Java bytecode into the machine language of the platform and runs it. JVMs can run on a client, in a browser, in a middle tier, on an intranet, on an application server, or in a database server.

**JAXB**
See Java Architecture for XML Binding.

**JAXP**
See Java API for XML Processing.

**JDBC**
See Java Database Connectivity.

**JDeveloper**
Oracle Java IDE that enables application, applet, and servlet development and includes an editor, compiler, debugger, syntax checker, help system, an integrated UML class modeler, and so on. JDeveloper has been enhanced to support XML-based development by including the Oracle Java XDK components, integrated for use along with XML support, in its editor.

**JDK**
See Java Developer’s Kit.

**JNDI**
See Java Naming and Directory Interface

**JSR**
See Java Specification Request

**JVM**
See Java virtual machine.
lazy type conversions
A mechanism used by Oracle XML DB to only convert the XML data for Java when the Java application first asks for it. This saves typical type conversion bottlenecks with JDBC.

listener
A separate application process that monitors the input process.

LOB
See large object.

marshalling
The process of traversing a Java content tree and writing an XML document that reflects the content of the tree. It is the inverse of unmarshalling.

name-level locking
Oracle XML DB provides for name-level locking rather than collection-level locking. When a name is added to a collection, an exclusive write lock is not placed on the collection, only that name within the collection is locked. The name modification is put on a queue, and the collection is locked and modified only at commit time.

node
In XML, the term used to denote each addressable entity in the DOM tree.

notation attribute declaration
In XML, the declaration of a content type that is not part of those understood by the parser. These types include audio, video, and other multimedia.

OAG
Open Applications Group.

OASIS
See Organization for the Advancement of Structured Information.

Object Request Broker (ORB)
Software that manages message communication between requesting programs on clients and between objects on servers. ORBs pass the action request and its parameters to the object and return the results back. Common implementations are JCORB and EJBs. See also CORBA.

OCT
See Ordered Collection in Tables.

OC4J
Oracle Containers for J2EE, a J2EE deployment tool that comes with JDeveloper.

Oracle Application Server
The Oracle Application Server product integrates all the core services and features required for building, deploying, and managing high-performance, n-tier, transaction-oriented Web applications within an open standards framework.
**ORACLE_HOME**
The operating system environment variable that identifies the location of the Oracle database installation for use by applications.

**Oracle CM SDK**
See Oracle Content Management Software Development Kit.

**Oracle Content Management SDK**
The Oracle file system and Java-based development environment that either runs inside the database or on a middle tier and provides a means of creating, storing, and managing multiple types of documents in a single database repository.

**Oracle XML Developer's Kit (XDK)**
The set of libraries, components, and utilities that provide software developers with the standards-based functionality to XML-enable their applications. In the case of the Oracle Java components of XDK, the kit contains an XML parser, an XSLT processor, the XML class generator, the JavaBeans, and the XSQL Servlet.

**Ordered Collection in Tables (OCT)**
When elements of a `VARRAY` are stored in a separate table, they are referred to as an Ordered Collection in Tables.

**Oracle Text**
An Oracle tool that provides full-text indexing of documents and the capability to do SQL queries over documents, along with XPath-like searching.

**Oracle XML DB**
A high-performance XML storage and retrieval technology provided with Oracle database server. It is based on the W3C XML data model.

**ORB**
See Object Request Broker.

**Organization for the Advancement of Structured Information (OASIS)**
An organization of members chartered with promoting public information standards through conferences, seminars, exhibits, and other educational events. XML is a standard that OASIS is actively promoting as it is doing with SGML.

**parent element**
An element that surrounds another element, which is referred to as its child element. For example, `<Parent><Child></Child></Parent>` illustrates a parent element wrapping its child element.

**Parsed Character Data (PCDATA)**
The element content consisting of text that should be parsed but is not part of a tag or nonparsed data.

**path name**
The name of a resource that reflects its location in the repository hierarchy. A path name is composed of a root element (the first `/`), element separators (`/`) and various sub-elements (or path elements). A path element may be composed of any character in the database character set except (`"", "/"`). These characters have a special meaning
for Oracle XML DB. Forward slash is the default name separator in a path name and backward slash may be used to escape characters.

**PCDATA**

See Parsed Character Data.

**PDA**

Personal Digital Assistant, such as a Palm Pilot.

**principal**

An entity that may be granted access control privileges to an Oracle XML DB resource. Oracle XML DB supports as principals:

- Database users.
- Database roles. A database role can be understood as a group, for example, the DBA role represents the DBA group of all the users granted the DBA role.

Users and roles imported from an LDAP server are also supported as a part of the database general authentication model.

**prolog**

The opening part of an XML document containing the XML declaration and any DTD or other declarations needed to process the document.

**PUBLIC**

The term used to specify the location on the Internet of the reference that follows.

**RDF**

Resource Definition Framework.

**renderer**

A software processor that produces a document in a specified format.

**repository**

The set of database objects, in any schema, that are mapped to path names. There is one root to the repository ("/") which contains a set of resources, each with a path name.

**resource**

An object in the repository hierarchy.

**resource name**

The name of a resource within its parent folder. Resource names must be unique (potentially subject to case-insensitivity) within a folder. Resource names are always in the UTF-8 character set (`NVARCHAR2`).

**result set**

The output of a SQL query consisting of one or more rows of data.

**root element**

The element that encloses all the other elements in an XML document and is between the optional prolog and epilog. An XML document is only permitted to have one root element.
SAX
See Simple API for XML.

schema
The definition of the structure and data types within a database. It can also be used to refer to an XML document that support the XML Schema W3C recommendation.

schema evolution
The process used to modify XML schemas that are registered with Oracle XML DB. Oracle XML DB provides the PL/SQL procedure `DBMS_XMLSCHEMA.CopyEvolve()`. This copies existing XML instance documents to temporary tables, drops and re-registers the XML schema with Oracle XML DB, and copies the XML instance documents to the new `XMLType` tables.

Secure Sockets Layer (SSL)
The primary security protocol on the Internet; it utilizes a public key/private key form of encryption between browsers and servers.

Server-Side Include (SSI)
The HTML command used to place data or other content into a Web page before sending it to the requesting browser.

servlet
A Java application that runs in a server, typically a Web or application server, and performs processing on that server. Servlets are the Java equivalent to CGI scripts.

SGML
See Structured Generalized Markup Language.

Simple API for XML (SAX)
An XML standard interface provided by XML parsers and used by event-based applications.

Simple Object Access Protocol (SOAP)
An XML-based protocol for exchanging information in a decentralized, distributed environment.

SOAP

SQL
See Structured Query Language.

SQL/XML
An ANSI specification for representing XML in SQL. Oracle SQL includes SQL/XML functions that query XML. The specification is not yet completed.

SSI
See Server-side Include.

SSL
See Secure Sockets Layer.
Structured Generalized Markup Language (SGML)
An ISO standard for defining the format of a text document implemented using markup and DTDs.

Structured Query Language (SQL)
The standard language used to access and process data in a relational database.

stylesheet
In XML, the term used to describe an XML document that consists of XSL processing instructions used by an XSLT processor to transform or format an input XML document into an output one.

SYSTEM
Specifies the location on the host operating system of the reference that follows.

SYS_XMLAGG
The native SQL function that returns as a single XML document the results of a passed-in SYS_XMLGEN SQL query. This can also be used to instantiate an XMLType.

SYS_XMLGEN
The native SQL function that returns as an XML document the results of a passed-in SQL query. This can also be used to instantiate an XMLType.

tag
A single piece of XML markup that delimits the start or end of an element. Tags start with < and end with >. In XML, there are start-tags (<name>), end-tags (</name>), and empty tags (<name/>).

TransX Utility
TransX Utility is a Java API that simplifies the loading of translated seed data and messages into a database.

UDDI
See Universal Description, Discovery and Integration.

UIX
See User Interface XML.

Uniform Resource Identifier (URI)
The address syntax that is used to create URLs and XPaths.

Uniform Resource Locator (URL)
The address that defines the location and route to a file on the Internet. URLs are used by browsers to navigate the World Wide Web and consist of a protocol prefix, port number, domain name, directory and subdirectory names, and the file name. For example, http://www.oracle.com:80/technology/tech/xml/index.htm specifies the location and path a browser will travel to find the OTN XML site on the World Wide Web.

Universal Description, Discovery and Integration (UDDI)
This specification provides a platform-independent framework using XML to describe services, discover businesses, and integrate business services on the Internet.
unmarshalling
The process of reading an XML document and constructing a tree of Java content objects. Each content object corresponds directly to an instance in the input document of the corresponding schema component.

See Also: marshalling

URI
See Uniform Resource Identifier.

URL
See Uniform Resource Locator.

User Interface XML (UIX)
A set of technologies that constitute a framework for building Web applications.

valid
The term used to refer to an XML document when its structure and element content is consistent with that declared in its associated DTD or XML schema.

W3C
See World Wide Web Consortium (W3C).

WebDAV
See World Wide Web distributed authoring and versioning.

Web Request Broker
The cartridge within Oracle Application Server that processes URLs and sends them to the appropriate cartridge.

Web Services Description Language (WSDL)
A general purpose XML language for describing the interface, protocol bindings, and deployment details of Web services.

well-formed
The term used to refer to an XML document that conforms to the syntax of the XML version declared in its XML declaration. This includes having a single root element, properly nested tags, and so forth.

Working Group (WG)
The committee within the W3C that is made up of industry members that implement the recommendation process in specific Internet technology areas.

World Wide Web Consortium (W3C)

World Wide Web Distributed Authoring and Versioning (WebDAV)
The Internet Engineering Task Force (IETF) standard for collaborative authoring on the Web. Oracle XML DB Foldering and Security features are WebDAV-compliant.

WSDL
See Web Services Description Language.
**World Wide Web**
A worldwide hypertext system that uses the Internet and the HTTP protocol.

**XDBbinary**
An XML element defined by the Oracle XML DB schema that contains binary data. XDBbinary elements are stored in the repository when completely unstructured binary data is uploaded into Oracle XML DB.

**XDK**
See Oracle XML Developer's Kit.

**XLink**
The XML Linking language consisting of the rules governing the use of hyperlinks in XML documents. These rules are being developed by the XML Linking Group under the W3C recommendation process. This is one of the three languages XML supports to manage document presentation and hyperlinks (XLink, XPointer, and XPath).

**XML**
See eXtensible Markup Language.

**XML Base**
A W3C recommendation that describes the use of the xml:base attribute, which can be inserted in an XML document to specify a base URI other than the base URI of the document or external entity. The URIs in the document are resolved by means of the given base.

**XML Gateway**
A set of services that allows for integration with the Oracle E-Business Suite to create and consume XML messages triggered by business events.

**XML Namespaces**
The term to describe a set of related element names or attributes within an XML document. The namespace syntax and its usage is defined by a W3C Recommendation. For example, the `<xsl:apply-templates/>` element is identified as part of the XSL namespace. Namespaces are declared in the XML document or DTD before they are used, with the following attribute syntax:

```
xmns:xsl=http://www.w3.org/TR/WD-xsl*
```

**XML Parser**
In XML, a software program that receives an XML document and determines whether it is well-formed and, optionally, valid. The Oracle XML parser supports both SAX and DOM interfaces.

**XML Pipeline Definition Language**
W3C recommendation that enables you to describe the processing relations between XML resources.

**XML processor**
A software program that reads an XML document and processes it, that is, performs actions on the document based on a set of rules. Validity checkers and XML editors are examples of processors.
XML Query (XQuery)
The on-going effort of the W3C to create a standard for the language and syntax to query XML documents.

XML schema
A document written in the XML Schema language.

XML Schema
See XML Schema language.

XML Schema Definition
Equivalent to XML Schema language.

XML Schema language
The XML Schema language, also called simply "XML Schema," is a W3C standard for the use of simple data types and complex structures within an XML document. It addresses areas currently lacking in DTDs, including the definition and validation of data types.

Oracle XML Schema processor automatically ensures validity of XML documents and data used in e-business applications, including online exchanges. It adds simple and complex datatypes to XML documents and replaces DTD functionality with an XML schema definition XML document.

XMLSchema-instance mechanism
Allows Oracle XML DB protocol servers to recognize that an XML document inserted into Oracle XML DB repository is an instance of a registered XML schema. This means that the content of the instance document is automatically stored in the default table defined by that XML schema. Defined by the W3C XML Schema working group and based on adding attributes that identify the target XML schema to the root element of the instance document. These attributes are defined by the XMLSchema-instance namespace.

XMLSchema-instance namespace
Used to identify an instance document as a member of the class defined by a particular XML schema. You must declare the XMLSchema-instance namespace by adding a namespace declaration to the root element of the instance document. For example: xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance.

XML schema registration
When using Oracle XML DB, you must first register your XML schema. You can then use the XML schema URLs while creating XMLType tables, columns, and views.

XML SQL Utility (XSU)
This Oracle utility can generate an XML document (string or DOM) given a SQL query or a JDBC ResultSet object. XSU can also extract the data from an XML document, then insert, update, or delete rows in a database table.

XMLType
XMLType is an Oracle datatype that stores XML data using an underlying CLOB column, or object-relational columns, or a binary format within a table or view.
XMLType views
Oracle XML DB provides a way to wrap existing relational and object-relational data in XML format. This is especially useful if, for example, your legacy data is not in XML but you have to migrate it to an XML format.

XPath
The open standard syntax for addressing elements within a document used by XSL and XPointer. XPath is currently a W3C recommendation. It specifies the data model and grammar for navigating an XML document utilized by XSLT, XLink and XML Query.

XPath rewrite
Can be used when the XMLType is stored in structured storage (object-relational) using an XML schema. Queries using XPath can potentially be rewritten directly to underlying object-relational columns. XPath query rewrite is used for XPaths in SQL functions such as existsNode(), extract(), extractValue(), and updateXML(). It enables the XPath to be evaluated against the XML document without constructing the XML document in memory.

XPointer
The term and W3C recommendation to describe a reference to an XML document fragment. An XPointer can be used at the end of an XPath-formatted URI. It specifies the identification of individual entities or fragments within an XML document using XPath navigation.

XSL
See eXtensible Stylesheet Language.

XSLFO
See eXtensible Stylesheet Language Formatting Object.

XSLT
See eXtensible Stylesheet Language Transformation.

XSLT Virtual Machine (XVM)
Oracle’s XSLT Virtual Machine is the software implementation of a "CPU" designed to run compiled XSLT code. The concept of virtual machine assumes a compiler compiling XSLT stylesheets to a program of byte-codes, or machine instructions for the "XSLT CPU".

XSQL pages
XML pages that contain instructions for the XSQL servlet.

XSQL servlet
A Java-based servlet that can dynamically generate XML documents from one or more SQL queries and optionally transform the documents in the server with an XSLT stylesheet.

XSU
See XML SQL Utility.
Symbols
<xsql:dml> action, 30-6, 30-8, 30-10, 30-11, 30-24, 30-40
<xsql:include-owa> action, 30-13
<xsql:include-param> action, 30-15
<xsql:include-posted-xml> action, 30-16
<xsql:include-request-params> action, 30-17
<xsql:include-xml> action, 30-19, 30-20, 30-22
<xsql:query> action, 30-26
<xsql:set-cookie> action, 30-31
<xsql:set-page-param> action, 30-33
<xsql:set-session-param> action, 30-36
<xsql:set-stylesheet-param> action, 30-38

A
access control entry, definition, Glossary-1
access control list, definition, Glossary-1
ACE, definition, Glossary-1
ACL, definition, Glossary-1
application server, definition, Glossary-1
attribute, definition, Glossary-1

B
B2B, definition, Glossary-1
B2C, definition, Glossary-1
Binary XML
   saving text as, 4-20
binary XML
   C, 19-1
decoder, 5-7
decoding, 5-4
encoder, 5-7
encoding, 5-4
Glossary, 5-2
models for using, 5-2
using Java, 5-6
vocabulary management, 5-5
XML DB, 5-6
binary XML for Java, 5-1
Binary XML Storage Format, 5-1
binding
clearBindValues(), 11-6
Built-in Action Handler, 15-21
Built-in Action Handler, XSQL, 15-21
Business-to-Business, definition, Glossary-1
Business-to-Consumer, definition, Glossary-1

C
C API, 16-10
C compile-time environment on UNIX
   setting up, 16-4
C components
demos, 16-1, 17-6, 18-7, 20-3
directory structure, 16-1
globalization support, 16-11
installation, 16-1
runtime environment on Windows, 16-6
   samples, 16-1, 17-6, 18-7, 20-3
   setting up Windows environment, 16-5
   setting up Windows environment variables, 16-6
   with Visual C/C++ on Windows, 16-8
C environment variables on UNIX, 16-3
C libraries
   contents, 16-2
   runtime environment on UNIX, 16-3
C++ class generator, 1-6
C++ interface, 24-1
callback, definition, Glossary-2
cartridge, definition, Glossary-2
Cascading Style Sheets, definition, Glossary-2, Glossary-3
CDATA, definition, Glossary-2
Class Generator
   XML C++, 29-1
Class Generator, definition, Glossary-2
CLASSPATH
   XSQL Pages, 14-6
CLASSPATH, definition, Glossary-2
clearBindValues(), 11-6
clearUpdateColumnNames(), 11-33
command-line interface
   oraxml, 4-13, 8-8
Common Object Request Broker API,
definition, Glossary-2
Common Oracle Runtime Environment,
definition, Glossary-2
Connection Definitions, 14-7
context, creating one in XSU PL/SQL API, 11-37
CORBA, definition, Glossary-2
CORE, definition, Glossary-2
creating context handles
    getCtx, 11-6
custom connection manager, 15-25

D
DAD, definition, Glossary-3
Data Provider for .NET, 1-16
data variables into XML, 4-46
Database Access Descriptor, definition, Glossary-3
datagram, definition, Glossary-3
DB Access JavaBean, 10-3
DBMS.XMLQuery
    clearBindValues(), 11-6
getXMLClob, 11-7
DBMS.XMLQuery(), 11-6
DBMS.XMLSave, 11-7
deleteXML, 11-8
getCtx, 11-7
insertXML, 11-8
updateXML, 11-8
DBMS.XMLSave(), 11-7
DBURITYPE, definition, Glossary-3
decoding binary XML, 5-4
Default SQL to XML Mapping, 11-37
demos
    C components, 16-1, 17-6, 18-7, 20-3
directory structure
    C, 16-1
DOCTYPE, definition, Glossary-3
document creation Java APIs, 2-3
document Location Hint, definition, Glossary-3
document Object Model, definition, Glossary-3
document Type Definition, definition, Glossary-3
DOM
    creating in Java, 4-1
    specifications, 31-2
DOM fidelity, definition, Glossary-4
DOM, definition, Glossary-3
DOMBuilder Bean, 10-2
DTD, definition, Glossary-3
DTDs
    external, 4-47

E
EDI, definition, Glossary-4
EJB, definition, Glossary-4
Electronic Data Interchange, definition, Glossary-4
element, definition, Glossary-4
empty element, definition, Glossary-4
encoding binary XML, 5-4
Enterprise JavaBean, definition, Glossary-4
entity, definition, Glossary-4
examples of document creation in Java, 2-3
eXtensible Stylesheet Language Formatting Object,
    definition, Glossary-5
eXtensible Stylesheet Language Transformation,
    definition, Glossary-5
eXtensible Stylesheet Language,
    definition, Glossary-4

F
FileReader not for system files, 4-50
FOP
    serializer, 14-9
    serializer to produce PDF, 15-16
FOP, definition, Glossary-5

G
generated XML
    customizing, 11-40
generating XML, 11-16
    using DBMS.XMLQuery, 11-6
    using XSU command line, getXML, 11-16
getCtx, 11-6, 11-7
getXML, 11-16
getXMLClob, 11-7
getXMLClob, 11-7
globalization support
    for the C components, 16-11

H
HASPATH, definition, Glossary-5
hierarchical indexing, definition, Glossary-5
HTML Form Parameters, 14-26
HTTP Parameters, 14-24
HTTP POST method, 14-29
HTTPURITYPE, definition, Glossary-5

I
IDE, definition, Glossary-6
IIOP, definition, Glossary-6
infoset, 5-3
infoset, definition, Glossary-5
INPATH, definition, Glossary-5
insert, XSU, 11-42
insertXML, 11-8
installation
    C components, 16-1
instance document, definition, Glossary-5
instantiate, definition, Glossary-6
Integrated Development Environment,
    definition, Glossary-6
invalid characters, 4-52

J
JAR files, DTDs, 4-47
Java 2 Platform, Enterprise Edition,
    definition, Glossary-6
Java API for XML Processing (JAXP),
    definition, Glossary-6
Java Architecture for XML Binding (JAXB),
    definition, Glossary-6
Java classes deprecated, 2-2
Java components
  creating a DOM, 4-1
  environment in Windows, 3-6
  installation, 3-1
  parsing, 4-1
Java Database Connectivity, definition, Glossary-6
Java Naming and Directory Interface, definition, Glossary-7
Java Runtime Environment, definition, Glossary-7
Java, definition, Glossary-6
JavaBean, definition, Glossary-6
JAXB
  class generator, 1-6
  compared with JAXP, 8-1, 8-4
  features not supported, 8-9
  marshalling and unmarshalling, 8-1
  validating, 8-1
  what is, 8-4
JAXP
  compared with JAXB, 8-1
JAXP (Java API for XML Processing), 4-37
JCR 1.0 standard, 4-2
JDBC driver, 11-3
JDBC, definition, Glossary-6, Glossary-7
JDeveloper, 1-14
JDeveloper, definition, Glossary-7
JDK, definition, Glossary-6
JNDI, definition, Glossary-7
JRE, definition, Glossary-7
JSP, definition, Glossary-7
JSR 170 standard, 4-2
JSR, definition, Glossary-7
JVM, definition, Glossary-7

L
lazy type conversions, definition, Glossary-8
listener, definition, Glossary-8

M
make.bat file
  editing on Window for C environment, 16-7
mapping
  primer, XSU, 11-37
method
  getDocument(), DOMBuilder Bean, 10-6
methods
  addXSLTransformerListener(), 10-8
  domBuilderError(), 10-6
  DOMBuilderOver(), 10-6, 10-8
  domBuilderStarted(), 10-6
Microsoft .NET, 1-16
Multimedia, definition, Glossary-6

N
name-level locking, definition, Glossary-8
namespace, definition, Glossary-8
.NET, 1-16

O
OAG, definition, Glossary-8
OASIS, definition, Glossary-9
OC4J, definition, Glossary-8
OCI and the XDK C, 18-17
OCI examples, 18-19
Open Applications Group, definition, Glossary-8
Oracle Content Management SDK, definition, Glossary-9
Oracle Text, definition, Glossary-9
Oracle XML DB, definition, Glossary-9
ORACLE_HOME, definition, Glossary-9
OracleXMLXSU command line, 11-14
OracleXml namespace, 24-2
OracleXMLSQLException, 11-29
oraxml, 4-13, 8-8
oraxsl
  command line interfaces, 6-6
  supported database, 4-10
using DTDs, 4-47
Parser for Java, overview, 4-9
path name, definition, Glossary-9
PCDATA, definition, Glossary-9
PDA, definition, Glossary-9
PDF results using FOP, 14-9
Personal Digital Assistant, definition, Glossary-10
Pipeline Definition Language, 9-1
Pipeline Definition Language, definition, Glossary-14
PL/SQL
  generating XML with DBMS_XMLQuery, 11-6
principal, definition, Glossary-10
prolog, definition, Glossary-10
PUBLIC, definition, Glossary-10

R
renderer, definition, Glossary-10
Reports, Oracle, 1-16
repository, definition, Glossary-10
Resource Definition Framework, definition, Glossary-10
resource name, definition, Glossary-10
result set, definition, Glossary-10
root element, definition, Glossary-10

S
samples
  C components, 16-1, 17-6, 18-7, 20-3
SAX, definition, Glossary-11
schema evolution, definition, Glossary-11
schema, definition, Glossary-11
Secure Sockets Layer, definition, Glossary-11
security, XSQL Pages, 14-29
select
  with XSU, 11-42
Server-Side Include (SSI), definition, Glossary-11
servlet, definition, Glossary-11
servlet, XSQL, 14-1, 15-1
setMaxRows(), 11-31
setMaxRows(), 11-31
setStyleSheetHeader(), 11-31
setUpdateColumnHeader(), 11-33
setUpdateColumnNames() XML SQL Utility (XSU)
  setUpdateColumnNames(), 11-13, 11-26
setXSLT(), 11-31
SGML, definition, Glossary-12
Simple API for XML, definition, Glossary-11
Simple Object Access Protocol (SOAP), definition, Glossary-11
SOAP
  C clients, 22-4
  C examples, 22-6
  C Functions, 22-5
  for C, 22-1
  server, 22-4
  what is, 22-2
SOAP, definition, Glossary-11
SQL, definition, Glossary-12
SQL/XML, definition, Glossary-11
SSI, definition, Glossary-11
storing XML in the database, 11-7
streaming validator, 20-4
  opaque mode, 20-6
  transparent mode, 20-4
string data, 4-52
stylesheet, definition, Glossary-12
SYS_XMLAGG, definition, Glossary-12
SYS_XMLGEN, definition, Glossary-12
SYSTEM, definition, Glossary-12

T
tag, definition, Glossary-12
TransX Utility, 12-1

U
UIX, 1-16
UIX, definition, Glossary-13
Unicode in a system file, 4-50
unified C API for XDK and XML DB, 16-10
Unified Java API, 2-1
unified Java API, 2-1
Unified Java API new objects and methods, 2-3
Uniform Resource Identifier, definition, Glossary-12
Uniform Resource Locator, definition, Glossary-12
UNIX environment for C components
  configuring, 16-2
  update, XSU, 11-43
URI, definition, Glossary-12
URL, definition, Glossary-12
User Interface XML, 1-16
User Interface XML (UIX), definition, Glossary-13
UTF-16 Encoding, 4-52
UTF-8 output, 4-51

V
valid, definition, Glossary-13
validation
  auto validation mode, 4-8
  DTD validating Mode, 4-8
  partial validation mode, 4-8
  schema validation, 4-8
  schema validation mode, 4-8
Visual C/C++, 16-8
Visual Studio, 16-8

W
W3C, definition, Glossary-13
Web Request Broker, definition, Glossary-13
WebDAV, definition, Glossary-13
well-formed, definition, Glossary-13
WG, definition, Glossary-13
Windows, 16-5
  C components
    with Visual C/C++, 16-8
  C libraries, 16-5
  editing make.bat file, 16-7
  setting up C environment variables, 16-6
Windows environment for C components
  setting up, 16-5
WML Document, 14-25
World Wide Web Consortium, definition, Glossary-13
World Wide Web Distributed Authoring and Versioning, definition, Glossary-13
World Wide Web, definition, Glossary-14
WRB, definition, Glossary-13

X
XDBbinary, definition, Glossary-14
Xdiff instance document, 21-4
Xdiff schema, 21-6
XDK components, 1-1
XDK version
  using C, 16-5
  using Java, 3-8, 23-2
XDK, definition, Glossary-9
XLink, definition, Glossary-14
XML Base, 31-1
XML Base, definition, Glossary-14
XML C++ Class Generator, 29-1
XML Developer's Kit (XDK), definition, Glossary-9
XML documents
  generating from C, 1-12
  generating from C++, 1-12
  generating from Java, 1-10
XML Gateway, 1-16
XML Gateway, definition, Glossary-14
XML Namespaces 1.0, 31-1
XML output in UTF-8, 4-51
XML parser
  oraxml command-line interface, 4-13, 8-8
XML parser for C
  sample programs, 17-6, 18-7
XML parser, definition, Glossary-14
XML pull parser
  example, 18-15
XML Pull Parser error handling, 18-15
XML Pull Parser for C, 18-12
XML Query, definition, Glossary-15
XML Schema
  explained, 7-3
  processor for Java
    how to run the sample program, 6-4, 7-9, 8-7, 9-6, 12-5
XML schema for C
  sample programs, 20-3
XML schema registration, definition, Glossary-15
XML Schema, definition, Glossary-15
XML SQL Utility, Glossary-15
XML SQL Utility (XSU), 1-7
advanced techniques, exception handling
  (PL/SQL), 11-36
clearBindValues() with PL/SQL API, 11-6
connecting with OCI+ JDBC driver, 11-3
creating context handles with getCtx, 11-6
customizing generated XML, 11-40
DBMS_XMLQuery, 11-6
DBMS_XMLSave(), 11-7
dependencies and installation, 11-2
explained, 11-2
generateXML line, 11-16
generateXMLClob, 11-7
inserts, 11-42
mapping primer, 11-37
selects, 11-42
setKeyColumn() function, 11-28
setRaiseNoRowsException(), 11-31
updates, 11-43
XML SQL Utility (XSU)
  setXSLT(), 11-31
XML, definition, Glossary-4
xmlcq usage, 29-2
XMLCompress JavaBean, 10-4
XMLDBAccess JavaBean, 10-3
XmlDiff command line utility, 21-2
XmlDiff example, 21-8
XmlDiff in C, 21-1
XmlDiff JavaBean, 10-3
XMLGEN, is obsolete. See DBMS_XMLQUERY and DBMS_XMLSAVE, 11-2
XmHash, 21-12
XmlNode.selectNodes() method, 4-45
Xmpatch, 21-11
XMLSchema-instance mechanism, definition, Glossary-15
XMLSchema-instance namespace, definition, Glossary-15
XMLSchema-instance mechanism, definition, Glossary-14
XMLType views, definition, Glossary-16
XPath rewrite, definition, Glossary-16
XPath, definition, Glossary-16
XPointer, definition, Glossary-16
XSDBuilder, 4-8
XSL Transformation (XSLT) Processor, 1-5
XSL Transformation (XSLT) Processor for Java, 6-3, 9-3, 12-3
XSL Transformations Specifications, 31-3
XSL, definition, Glossary-4
XSLFO, definition, Glossary-5
XSLT
  XSLTransformer bean, 10-7
XSLT compiler, 17-1
XSLT processor, 17-3
XSLT Processor for Java
  hints for using, 6-13
XSLT stylesheets
  setStylesheetHeader() in XSU PL/SQL, 11-31
  setXSLT() with XSU PL/SQL, 11-31
XSLT, definition, Glossary-5
XSLTransformer JavaBean, 10-2
XSLValidator JavaBean, 10-4
XSQL
  action handler errors, 15-11
  advanced topics, 15-1
  built-in action handler elements, 15-21
  connection, 14-28
  current page name, 14-28
  errors, 14-29
  setting up demos, 14-9, 14-10
  SOAP support, 14-28
  stylesheets, 15-2
  two queries, 14-25
XSQL Pages security, 14-29
XSQL servlet
  hints, 14-25
  XML Servlet examples, 14-8
XSQL, definition, Glossary-16
XSU
  generating XML, 11-16
  mapping primer, 11-37
usage guidelines, 11-37
XSU (XML SQL Utility), 1-7
XSU usage techniques, 11-37
XVM
   XSLT compiler, 17-3
XVM (XSLT Virtual Machine) processor, 17-1
XVM, definition, Glossary-16