Oracle® XML Developer's Kit

Programmer's Guide

10g Release 2 (10.1.2)

Part No. B14033-01

June 2005

ORACLE®

Oracle XML Developer's Kit Programmer's Guide, 10g Release 2 (10.1.2)

Part No. B14033-01

Primary Author: Jack Melnick

Contributing Author: Mark Bauer, Shelley Higgins, Steve Muench, Mark Scardina, Jinyu Wang

Contributor: Lance Ashdown, Sandeepan Banerjee, Sivasankaran Chandrasekar, Dan Chiba, Steve Ding, Stanley Guan, Bill Han, K. Karun, Murali Krishnaprasad, Dmitry Lenkov, Roza Leyderman, Bruce Lowenthal, Ian Macky, Anjana Manian, Meghna Mehta, Valarie Moore, Ravi Murthy, Anguel Novoselsky, Tomas Saulys, Helen Slattery, Asha Tarachandani, Tim Yu, Jim Warner, Simon Wong, Kongyi Zhou

Contents

Send Us Your Comments

Oracle XML Developer's Kit Programmer's Guide, 10g Release 2 (10.1.2)

Part No. B14033-01

Oracle welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the title and part number of the documentation and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227. Attn: Server Technologies Documentation Manager
- Postal service:

  Oracle Corporation
  Server Technologies Documentation Manager
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and electronic mail address (optional).

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This Preface contains these topics:

- Audience
- Documentation Accessibility
- Structure
- Related Documents
- Conventions

Audience

Oracle XML Developer's Kit Programmer's Guide introduces application developers to the XML Developer's Kit (XDK) and how the various language components of the XDK can work together to generate and store XML data in a database or in a document outside the database. Examples and sample applications are introduced where possible.

To use this document, you need familiarity with XML and a third-generation programming language such as Java, C, or C++.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Structure

This document contains:

Chapter 1, "Overview of XML Developer's Kit Components"
Introduces the XDK parts and utilities used with them.

Chapter 2, "Getting Started with XDK Java Components"
How to install the XDK Java components.

Chapter 3, "XML Parser for Java"
Describes the XML parser for Java features.

Chapter 4, "XSLT Processor for Java"
Describes the XSLT processor for Java.

Chapter 5, "XML Schema Processor for Java"
Describes the XML schema processor Java.

Chapter 6, "Using JAXB Class Generator"
Describes JAXB, which replaces the XML class generator for Java.

Chapter 7, "XML SQL Utility (XSU)"
Describes the XML SQL utility for Java.

Chapter 8, "XSQL Pages Publishing Framework"
Describes this Java capability.

Chapter 9, "Pipeline Definition Language for Java"
Describes the implementation of the Pipeline Definition Language for Java.

Chapter 10, "XDK JavaBeans"
Describes the JavaBeans available.

Chapter 11, "Using XDK and SOAP"
A brief introduction to SOAP and the XDK.

Chapter 12, "TransX Utility"
The TransX Utility simplifies the loading of translated seed data and messages into a database.

Chapter 13, "Getting Started with XDK C Components"
How to install the XDK C components.

Chapter 14, "XML Parser for C"
You are requested to use the new unified C API for new XDK applications. The old C functions are supported only for backward compatibility, but will not be enhanced. Describes the C XML parser features.

Chapter 15, "XSLT Processors for C"
Describes the XSLT processor for C features.

Chapter 16, "XML Schema Processor for C"
Describes the XML schema processor for C features.

Chapter 17, "Getting Started with XDK C++ Components"
How to install the XDK C++ components.

Chapter 18, "Unified C++ Interfaces"
The unified C++ API is described. The interfaces are listed.

Chapter 19, "XML Parser for C++"
Describes the XML parser for C++ interfaces.

Chapter 20, "XSLT Processor for C++"
Describes the XSLT processor for C++ interfaces.

Chapter 21, "XML Schema Processor for C++"
Describes the XML schema processor for C++ interfaces.

Chapter 22, "XPath Processor for C++"
Describes the XPath C++ interfaces.

Chapter 23, "XML Class Generator for C++"
Describes the XML class generator for C++ features.

Chapter 24, "XSU for PL/SQL"
XML SQL Utility (XSU) PL/SQL API reflects the Java API in the generation and storage of XML documents from and to a database.

Glossary
Defines terms of interest to readers of this manual, and related XML manuals. If a term is used in this manual, a cross-reference to the definition is marked in bold.

Related Documents

For more information, see these Oracle resources:

- *Oracle XML DB Developer's Guide*
- *Oracle XML API Reference*
- *Oracle XML Java API Reference*

- *Oracle Streams Advanced Queuing User's Guide and Reference*
- 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:

- 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/
- *Oracle Database 10g XML & SQL* by the Oracle XML Product Development Team, http://www.osborne.com/oracle/
- *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, http://www.javaworld.com/jw-04-1999/jw-04-xml_p.html
- *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

- The XML FAQ by the W3C XML Special Interest Group (the industry clearing house for XML DTDs that allow companies to exchange XML data), `http://www.ucc.ie/xml/`
- `XML.org`, hosted by **OASIS** as a resource to developers of purpose-built XML languages, `http://xml.org/`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples
- Conventions for Windows Operating Systems

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, Recovery Manager keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executable programs, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names and connect identifiers, user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. *Note:* Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to start SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run *`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
`[ ]`	Anything enclosed in brackets is optional.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
`{ }`	Braces are used for grouping items.	`{ENABLE | DISABLE}`

|	A vertical bar represents a choice of two options.	`{ENABLE | DISABLE}`
		`[COMPRESS | NOCOMPRESS]`

`...`	Ellipsis points mean repetition in syntax descriptions.
	In addition, ellipsis points can mean an omission in code examples or text.
		`CREATE TABLE ... AS `*`subquery`*`;`
		`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`

Other symbols	You must use symbols other than brackets ([ ]), braces ({ }), vertical bars (|), and ellipsis points (...) exactly as shown.	`acctbal NUMBER(11,2);`
		`acct    CONSTANT NUMBER(4) := 3;`

`Italics`	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`*
		`DB_NAME = `*`database_name`*

`UPPERCASE`	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. Because these terms are not case sensitive, you can use them in either UPPERCASE or lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates user-defined programmatic elements, such as names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Conventions for Windows Operating Systems

The following table describes conventions for Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose **Start** > ***menu item***	How to start a program.	To start the Database Configuration Assistant, choose **Start** > **Programs** > **Oracle -** ***HOME_NAME*** > **Configuration and Migration Tools** > **Database Configuration Assistant**.

Convention	Meaning	Example
File and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the filename begins with \\, then Windows assumes it uses the Universal Naming Convention.	c:\winnt"\"system32 is the same as C:\WINNT\SYSTEM32
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`
Special characters	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (') do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	`C:\> exp HR/HR TABLES=emp QUERY=\"WHERE job='REP'\"`
`HOME_NAME`	Represents the Oracle home name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	`C:\> net start Oracle`*`HOME_NAME`*`TNSListener`

Convention	Meaning	Example
ORACLE_HOME and *ORACLE_BASE*	In releases prior to Oracle8*i* release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *ORACLE_HOME* directory. The default for Windows NT was `C:\orant`. This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *ORACLE_HOME* directory. There is a top level directory called *ORACLE_BASE* that by default is `C:\oracle\product\10.1.0`. If you install the latest Oracle release on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is `C:\oracle\product\10.1.0\db_`*n*, where *n* is the latest Oracle home number. The Oracle home directory is located directly under *ORACLE_BASE*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle Database Installation Guide for Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *ORACLE_BASE*`\`*ORACLE_HOME*`\rdbms\admin` directory.

What's New in Oracle XML Developer's Kit?

This section describes new features of the Oracle Database 10g Release 1 (10.1) and provides pointers to additional information. New features information from previous releases is also retained to help those users migrating to the current release.

The following sections describe the new features in Oracle XML Developer's Kit:

- Oracle Database 10g Release 1 (10.1) New Features in Oracle XML Developer's Kit

Oracle Database 10*g* Release 1 (10.1) New Features in Oracle XML Developer's Kit

This section contains new features in the Oracle XML Developer's Kit (XDK).

- JAXB Class Generator

  The JAXB compiler generates the interfaces and the implementation classes corresponding to the XML Schema. The JAXB Class Generator, which is based on the **Java Specification Request (JSR)** recommendation for JAXB, is to be used for new applications. The Class Generator for Java is deprecated and replaced by the JSR-31 implementation of XML Data Binding (JAXB). The runtime will be supported so that the Java classes generated in older releases continue to work.

  See Also: Chapter 6, "Using JAXB Class Generator"

- Unified API for C and C++

  These sets of functions work in both XDK and XML DB and replace the C and C++ XDK functions of previous releases.

  See Also: Chapter 14, "XML Parser for C" and Chapter 18, "Unified C++ Interfaces"

- XDK C/C++ Components Change

  Previously, the globalization support data environment variable setting was ORA_ NLS33. It has now been changed to ORA_NLS10.

- Pipeline Definition Language

  This W3C note is implemented for Java in the XDK.

  See Also: Chapter 9, "Pipeline Definition Language for Java"

- XSLT Compiler and XSLT Virtual Machine (XVM)

For improved performance there are new interfaces for the XSL processor for C and C++.

> **See Also:** "XVM Processor" on page 15-1.

- XSQL Pages Publishing Framework Updates

  This chapter has been updated for this release.

  > **See Also:** Chapter 8, "XSQL Pages Publishing Framework"

- Using SOAP

  New sections are included, plus an example of a SOAP project is found in this chapter:

  > **See Also:** Chapter 11, "Using XDK and SOAP"

- New XML JavaBeans

  XMLCompress, XMLDBAccess, and XSDValidator JavaBeans are now available in the XDK.

  > **See Also:** Chapter 10, "XDK JavaBeans"

- XDK Java Components Changes

  The XDK Java components in this release have several fixes for J2EE conformance and XML 1.0 Conformance Test Suite. Some of the changes resulted in change in behavior with respect to previous release. A few of them are listed below:

  - The default value of preserve whitewashes `[XMLParser.setPreserveWhitespace()]` is now dependent on the presence of a DTD. If a DTD is present, the default is `false`, else it is `true`. Earlier the default was always `false`.
  - `get Prefix()`,`getNamespaceURI()`,and `getLocalName()` return `null` instead of """ (empty string), when not present in the element or attribute, or if the node was created using DOM 1.0 methods.

  The PL/SQL wrapper for parsing and transformation is replaced by the DBMS_XMLPARSER, DBMS_XMLDOM and DBMS_XSLPROCESSOR packages.

  JAXP 1.2 supports XML Schema validation.

  `XMLSAXSerializer` provides support to handle the SAX output serialization.

  > **See Also:** "XDK Java Components Specifications" on page 2-1 for specifications of the levels of the components in this release

- Restructuring of XML Documentation

  The following PL/SQL chapters are now located in the *Oracle XML DB Developer's Guide*.

  - XML Parser for PL/SQL
  - XSLT Processor for PL/SQL
  - XML Schema Processor for PL/SQL

1 Overview of XML Developer's Kit Components

This chapter contains these topics:

Introducing Oracle XML Developer's Kit

Oracle **XML** Developer's Kit (XDK) is a set of components, tools, and utilities that eases the task of building and deploying XML-enabled applications.

Release notes for XDK are found in `/xdk/doc/readme.html`.

Table 1–1 summarizes the standards supported by the XDK Components:

Table 1–1 Standards Supported by XDK Components

Standard	Java	C	C++
XML 1.0 (Second Edition)	Full	Full	Full
XML Namespaces 1.0	Full	Full	Full
XML Base	Only in XSLT	Not supported	Not supported
XML Schema 1.0	Full	Full	Full
DOM 1.0	Full	Full	Full
DOM 2.0 Core	Full	Full	Full
DOM 2.0 Events	Full	Full	Full
DOM 2.0 Transversal and Range	Full	Full	Full
DOM 3.0 Load and Save (working draft)	Partial	Not supported	Not supported
DOM 3.0 Validation (working draft)	Full	Not supported	Not supported
SAX 1.0	Full	Full	Full
SAX 2.0 Core	Full	Full	Full
SAX 2.0 Extension	Full	Full	Full

Table 1–1 (Cont.) Standards Supported by XDK Components

Standard	Java	C	C++
XSLT 1.0	Full	Full	Full
XSLT 2.0 with backward compatibility	Partial support	Not supported	Not supported
XPath 1.0	Full	Full	Full
XPath 2.0 with backward compatibility (working draft)	Partial support	Not supported	Not supported
XML Pipeline 1.0 (Notes)	Partial	Not supported	Not supported
JAXP 1.1 (JSR Standard)	Full	Not applicable	Not applicable
JAXP 1.2 (JSR Standard)	Full	Not applicable	Not applicable
JAXB 1.0 (JSR Standard)	Partial support	Not applicable	Not applicable
Class Generator (Oracle proprietary product)	Not applicable	Not applicable	Full

Oracle has representatives participating actively in the following W3C Working Groups:

- XML Core
- XML Schema
- XML Query
- XSL/XPath
- XLink/XPointer
- XML Namespaces
- DOM
- SAX

Oracle has representatives participating actively on the following JSR standards:

- JAXB
- JAXP

Overview of Oracle XDK Components

The XDK is fully supported and comes with a commercial redistribution license. To provide a broad variety of deployment options, the XDK components are available for Java, C, and C++. Table 1–2 lists and describes the Oracle XDK components.

Table 1–2 Overview of Oracle XDK Components

XDK Component	Description	Supported Languages	See Also
XML Parsers	Create and parse XML using industry standard **DOM** and **SAX** interfaces.	Java, C, and C++	Chapter 3, "XML Parser for Java" Chapter 14, "XML Parser for C" Chapter 19, "XML Parser for C++"
XSLT Processors	Transforms or renders XML into other text-based formats such as **HTML**.	Java, C, and C++	Chapter 4, "XSLT Processor for Java" Chapter 15, "XSLT Processors for C" Chapter 20, "XSLT Processor for C++"
XVM	High performance XSLT transformation engine that supports compiled stylesheets.	C and C++	"XVM Processor" on page 15-1
XML Schema Processors	Validates schemas. It allows use of XML simple and complex datatypes.	Java, C, and C++	Chapter 5, "XML Schema Processor for Java" Chapter 16, "XML Schema Processor for C" Chapter 21, "XML Schema Processor for C++"
JAXP	Gives you the ability to use the SAX, DOM, and XSLT processors, or alternate processors, from your Java application.	Java	"Using JAXP" on page 3-36
JAXB Class Generator	Creates Java classes based on an XML Schema. Replaces XML Class Generator for Java.	Java	Chapter 6, "Using JAXB Class Generator"
XML **Class Generator**	Automatically generates C++ classes from **DTD**s and XML Schemas to send XML data from Web forms or applications.	C++	Chapter 23, "XML Class Generator for C++"
XML SQL Utility (XSU) (XSU)	Generates XML documents, DTDs and Schemas from **SQL** queries. Maps any SQL query result to XML and vice versa.	Java and PL/SQL	Chapter 7, "XML SQL Utility (XSU)" Chapter 24, "XSU for PL/SQL"
XSQL Servlet	Combines XML, SQL, and XSLT in the server to deliver dynamic Web content.	Java	Chapter 8, "XSQL Pages Publishing Framework"
XML **Pipeline Definition Language**	Applies a set of XML processes specified in a declarative XML `XPipe` file.	Java	Chapter 9, "Pipeline Definition Language for Java"
XML **JavaBeans**:	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, "XDK JavaBeans"

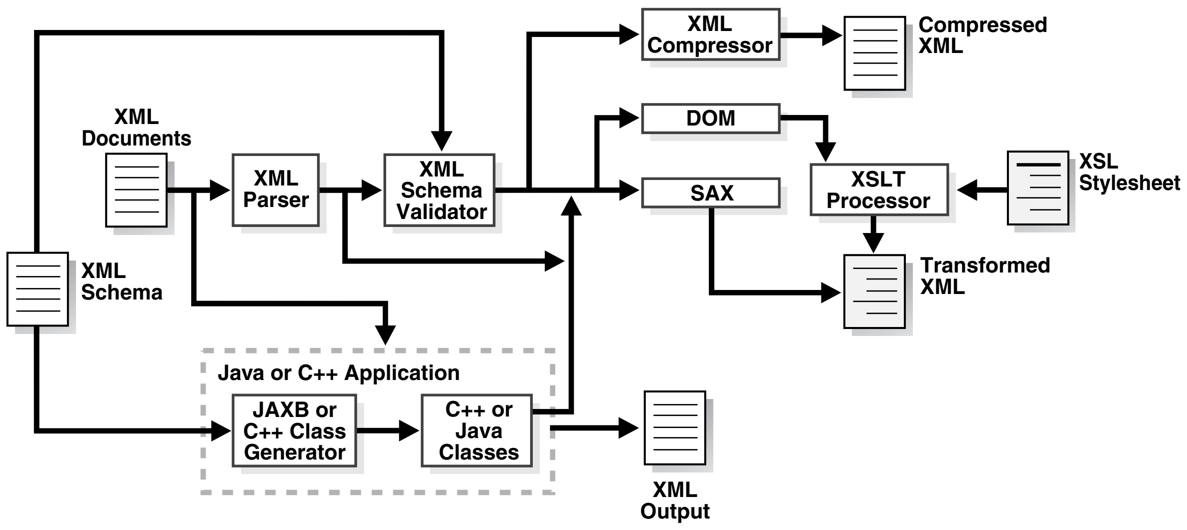
Table 1–2 (Cont.) Overview of Oracle XDK Components

XDK Component	Description	Supported Languages	See Also
Oracle **SOAP** Server	The Simple Object Access Protocol (SOAP) is a lightweight protocol for sending and receiving requests and responses across the Internet.	Java	Chapter 11, "Using XDK and SOAP"
TransX Utility	Loads translated seed data and messages into the database using XML.	Java	Chapter 12, "TransX Utility"
XML Compressor	Binary compression and decompression of XML documents.	Java	"About XML Compressor" on page 3-9

XDK Functionality Applied to XML Documents

To work with XML technology, you need to be familiar with the tools to parse XML, validate XML against a DTD or XML schema, transform XML by applying a stylesheet, and generate XML documents based on data selected from a database by means of SQL statements. See Table 1–2, " Overview of Oracle XDK Components", column "See Also," for cross-references to the components. XML Compressor supports only Java. Figure 1–1 shows a simple overview of XDK.

Figure 1–1 XDK Functionality

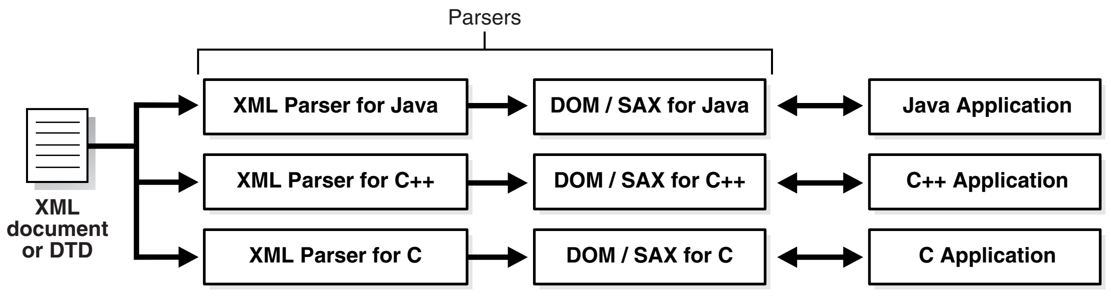


XML Parsers

The XML Parsers read the XML document, and use either DOM APIs for navigating a tree-like representation of the XML document, or a Simple API for XML (SAX) event-based interface that requires less memory. The XML Parser for Java supports JAXP. JAXP enables processing of XML documents using DOM, SAX, and XSLT independently of the XML processor implementation. XML Compressor is also integrated into the parser. This reduces the size of XML message payloads.

Figure 1–2 illustrates the Oracle XML Parsers functionality.

Figure 1–2 The XML Parsers: Java, C, C++



XSL Transformation (XSLT) Processors

The Oracle XSLT engine fully supports the W3C XSL Transformations recommendation. It has the following features:

Enables standards-based transformation of XML information inside and outside the database on any operating system.

The Oracle XML Parsers include an integrated XSL Transformation (XSLT) Processor for transforming XML data using XSL stylesheets. Using the XSLT processor, you can transform XML documents from XML to XML, to HTML, or to virtually any other text-based format.

See Also:

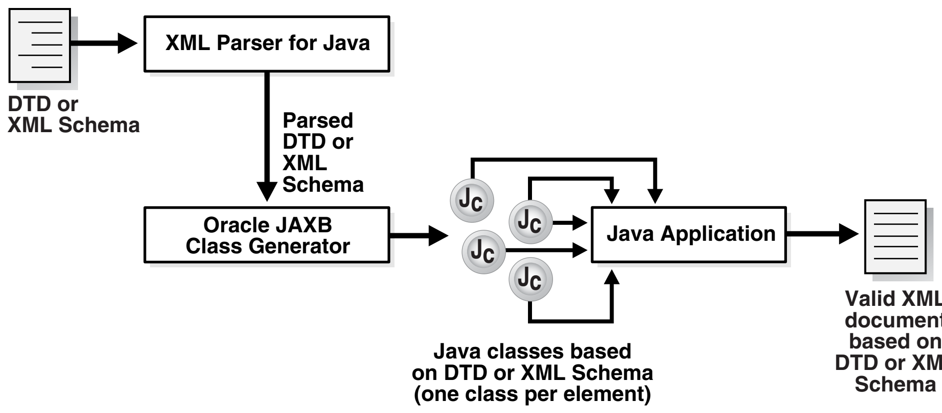
- "XSLT Processor for Java Overview" on page 4-1.
- Specifications and other information are found on the W3C site at `http://www.w3.org/Style/XSL`

JAXB and C++ Class Generators

JAXB Class Generator creates a set of Java classes for creation of XML documents corresponding to an input XML Schema. JAXB does not support DTDs. These classes are then used in a Java application. The C++ Class Generator creates a set of C++ classes for creation of XML documents corresponding to an input DTD or XML Schema. These classes are then used in a C++ application.

JAXB Class Generator supports data binding. An XML instance document can be input to load the instance data at runtime directly into the generated classes from a DTD. This improves memory usage and performance compared to DOM APIs.

Figure 1–3 Oracle JAXB Class Generator


XML Parser for Java
DTD or XML Schema
Parsed DTD or XML Schema
Oracle JAXB Class Generator
Jc
Java Application
Java classes based on DTD or XML Schema (one class per element)
Valid XML document based on DTD or XML Schema

XML Schema Processor

XML Schema was created by the W3C to describe the content and structure of XML documents in XML, thus improving on DTDs. XML Schema Processor introduces the concept of datatypes to XML. This allows data to be exchanged between databases using XML syntax.

XDK JavaBeans

The Oracle XDK JavaBeans are a set of visual and non-visual beans that are useful in creating a variety of XML-enabled Java applications or applets. XDK JavaBeans comprises the following beans:

DOMBuilder

The `DOMBuilder` JavaBean is a non-visual bean. It builds a DOM Tree from an XML document.

XSLTransformer

The `XSLTransformer` JavaBean is a non-visual bean. It accepts an XML file, applies the transformation specified by an input XSL stylesheet and creates the resulting output file.

DBAccess

`DBAccess` JavaBean maintains `CLOB` tables that contain multiple XML and text documents.

XMLDiff

The `XMLDiff` JavaBean performs a tree comparison on two XML DOM trees.

XMLCompress

This JavaBean is an encapsulation of the XML compression functionality.

XMLDBAccess

This JavaBean is an extension of the `DBAcess` bean to support the `XMLType` column, in which XML documents are stored in an Oracle Database table.

XSDValidator

This JavaBean is a class file that encapsulates the `oracle.xml.parser.schema.XSDValidator` class and adds capabilities for validating a DOM tree.

Oracle XML SQL Utility (XSU) for Java

XML SQL Utility is comprised of core Java class libraries that:

- Automatically and dynamically render the results of arbitrary SQL queries into canonical XML.
- Support queries over richly-structured user-defined object types and object views, including `XMLType`.

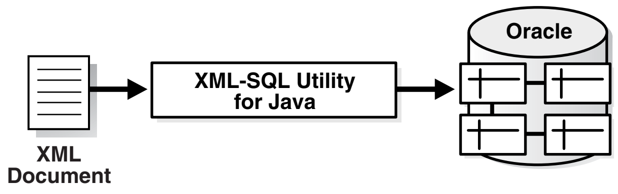
XML SQL Utility Java classes can be used for the following tasks:

- Load data from an XML document into an existing database schema or view.
- Support automatic XML insert of canonically-structured XML into any existing table, view, object table, or object view. By combining with XSLT transformations, virtually any XML document can be automatically inserted into the database.

Note: XSU for PL/SQL is also available. Oracle XML SQL Utility (XSU) for Java has these features.

Figure 1–4 shows the Oracle XML SQL Utility functionality for loading data from XML documents into a database schema or view:

Figure 1–4 Oracle XML SQL Utility Functional Diagram



XSQL Pages Publishing Framework

The XSQL Pages Publishing Framework (XSQL Servlet) is a server component that produces dynamic XML documents from one or more SQL queries of data objects. It does this by processing an XSQL file, which is simply an XML file with a specific structure and grammar. The XSQL Servlet uses Oracle's XML Parser to process this file and pass any XSLT processing statements to its internal XSLT Processor while passing the parameters and SQL statements between the tags to the XML SQL Utility. Results from those queries are then received as either XML-formatted text or a JDBC `ResultSet` object. If necessary, the query results can be further transformed into any desired format using the built-in XSLT processor.

TransX Utility

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

Soap Services

Oracle SOAP Services is published, located and executed through the Web. It is transport protocol-independent and operating system-independent. SOAP Services provide the standard XML message format for all applications. With SOAP Services, you can build messaging, RPC, and Web service applications with XML standards.

Using XDK-supported Languages to Generate XML Documents

Each of the language components will be employed to generate XML documents.

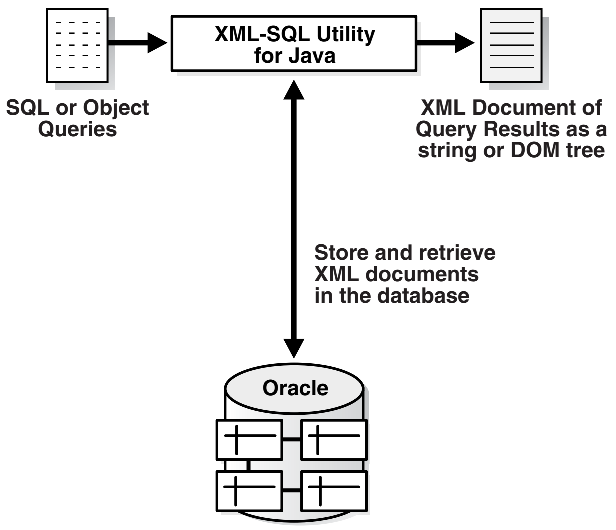
Using XSU for Java to Generate XML Documents

XSU can render the results of arbitrary SQL queries into canonical XML.

Generating XML from Query Results

Figure 1–5 shows how XML SQL Utility processes SQL queries and returns the results as an XML document.

Figure 1–5 XML SQL Utility Processes SQL Queries and Returns the Result as an XML Document


XML-SQL Utility for Java
SQL or Object Queries
XML Document of Query Results as a string or DOM tree
Store and retrieve XML documents in the database
Oracle

XML Document Structure: Columns Are Mapped to Elements

The structure of the resulting XML document has these attributes:

- 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

XSU Generates the XML Document as a String or DOM Element Tree

The XML SQL Utility (XSU) supports SAX event stream. XSU also generates either of the following:

- A string representation of the XML document. Use this representation if you are returning the XML document to a requester.
- An in-memory XML DOM tree of elements. Use this representation if you are operating on the XML programmatically, for example, transforming it using the XSLT Processor using DOM methods to search or modify the XML in some way.
- A series of SAX events which can be used when simply retrieving XML especially large documents or result sets.

XSU Generates a DTD Based on Queried Table's Schema

You can also use the XML SQL Utility (XSU) to generate a DTD or an XML Schema based on the schema of the underlying table or view being queried. You can use the generated DTD as input to the JAXB Class Generator for Java or the C++ Class Generator. This generates a set of classes based on the DTD elements. You can then write code that uses these classes to generate the infrastructure behind a Web-based form.

Based on this infrastructure, the Web form can capture user data and create an XML document compatible with the database schema. This data can then be written directly to the corresponding database table or object view without further processing.

See Also:

- Chapter 7, "XML SQL Utility (XSU)"
- "JAXB and C++ Class Generators" on page 1-5

Note: To write an XML document to a database table, where the XML data does not match the underlying table structure, transform the XML document before writing it to the database. For techniques on doing this, see Chapter 7, "XML SQL Utility (XSU)".

Using Oracle XDK Java Components to Generate XML Documents

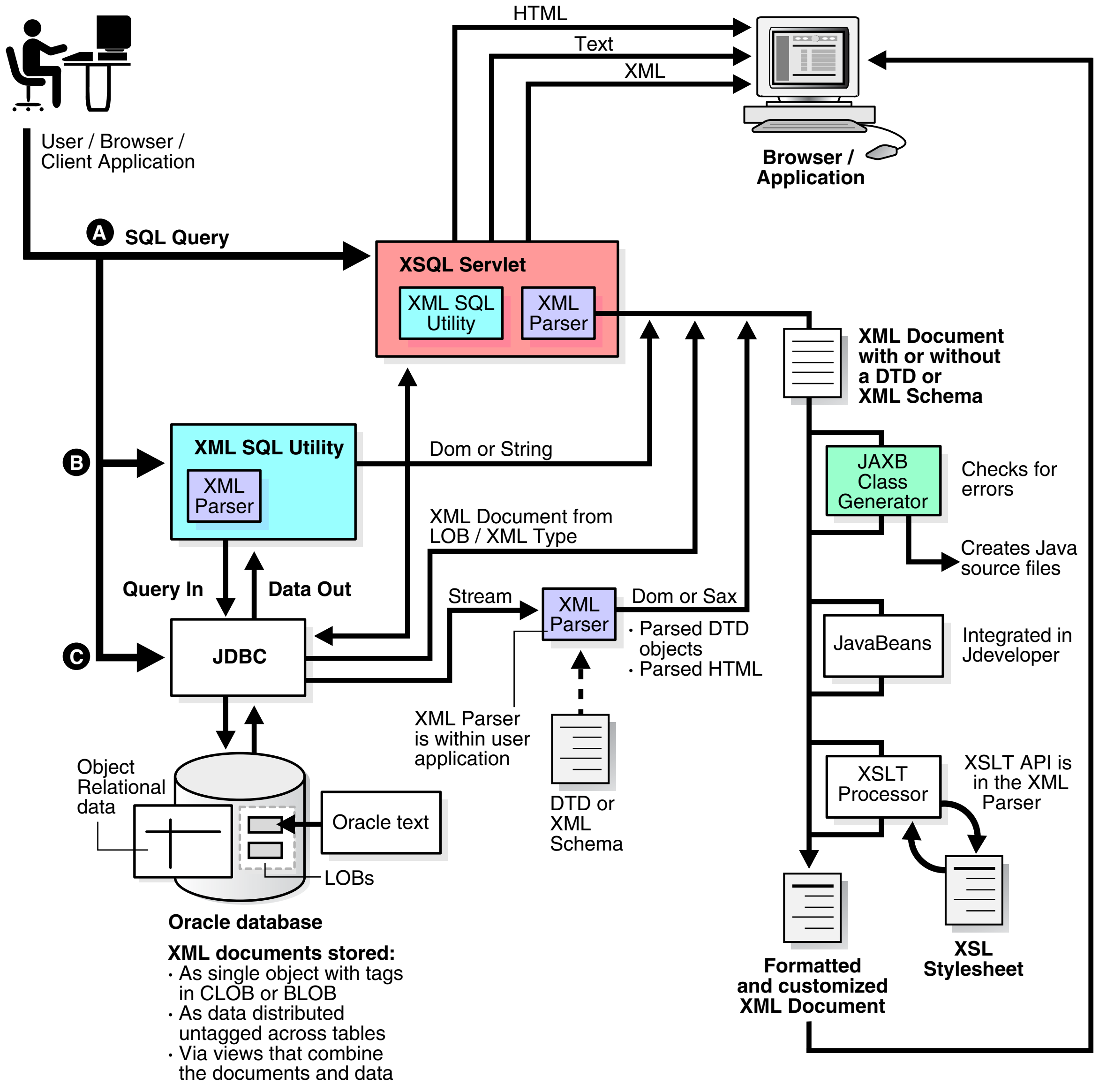
Figure 1–6 shows the Oracle XDK Java components and how they can be used to generate XML documents. Cross-references to XDK Java components are listed in Table 1–2, " Overview of Oracle XDK Components".

In the Java environment, when a SQL query is sent, these are the possible ways of processing the query using the Oracle XDK components:

- By the XSQL Servlet (this includes using XSU and XML Parser for Java)
- Directly by the XSU (this includes XML Parser for Java)
- Directly by JDBC which then accesses XML Parser

Regardless of which way the stored XML data is generated from the database, the resulting XML document output from the XML Parser for Java is further processed; it is formatted and customized by applying stylesheets and processed by the XSLT.

Figure 1–6 Generating XML Documents Using XDK Java Components



Using Oracle XDK C Components to Generate XML Documents

Figure 1–7 shows the Oracle XDK C language components used to generate XML documents. Available XDK C components are listed in Table 1–2, " Overview of Oracle XDK Components"

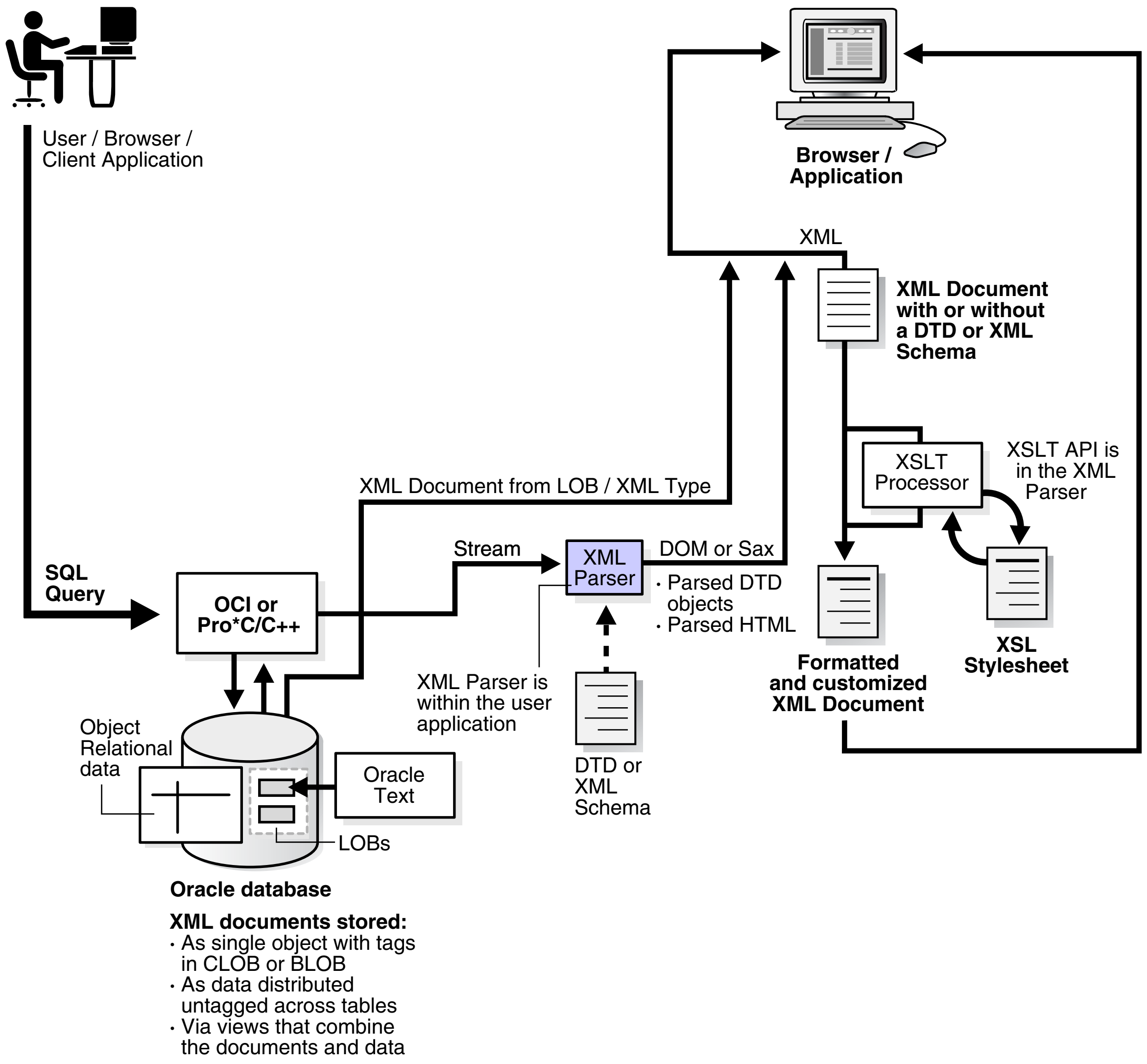
SQL queries can be sent to the database by Oracle Call Interface (OCI) or by the Pro*C/C++ Precompiler.

The resulting XML data can be processed in the following ways:

- With the XML Parser
- From the CLOB as an XML document

This XML data is optionally transformed by the XSLT processor, viewed directly by an XML-enabled browser, or sent for further processing to an application.

Figure 1–7 Generating XML Documents Using XDK C Components


User / Browser / Client Application
Browser / Application
XML
XML Document with or without a DTD or XML Schema
XSLT Processor
XSLT API is in the XML Parser
XML Document from LOB / XML Type
Stream
XML Parser
DOM or Sax
· Parsed DTD objects
· Parsed HTML
SQL Query
OCI or Pro*C/C++
Formatted and customized XML Document
XSL Stylesheet
XML Parser is within the user application
Object Relational data
Oracle Text
DTD or XML Schema
LOBs
Oracle database
XML documents stored:
· As single object with tags in CLOB or BLOB
· As data distributed untagged across tables
· Via views that combine the documents and data

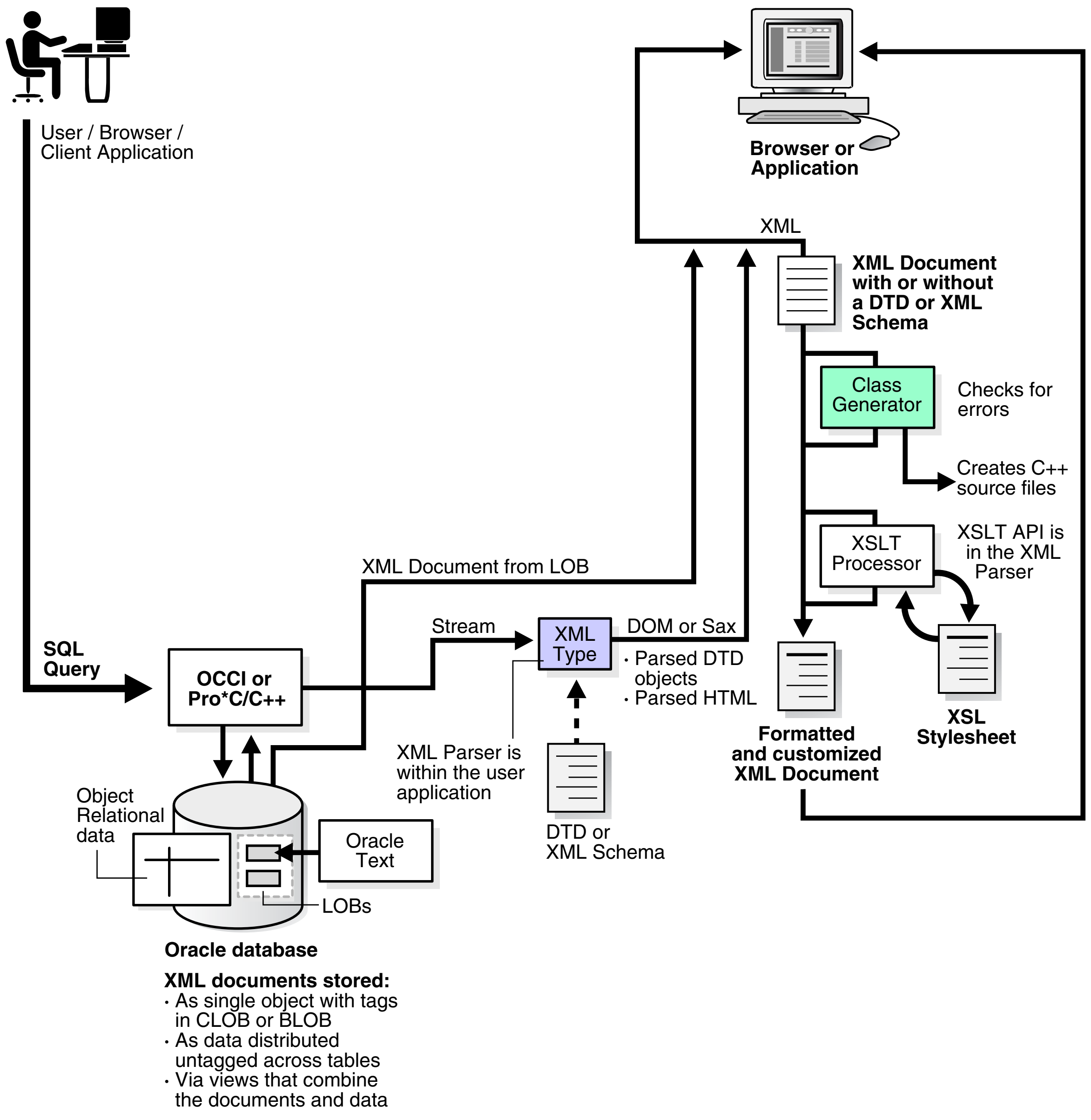
Using Oracle XDK C++ Components to Generate XML Documents

Figure 1–8 shows the Oracle XDK C++ components used to generate XML documents. Available XDK C++ components are listed in Table 1–2, " Overview of Oracle XDK Components"

In the C++ environment, when a user or client or application sends a SQL query, there are two possible ways of processing the query using the XDK C++ components:

- Directly by JDBC which then accesses the XML Parser
- Through the Oracle C++ Call Interface (OCCI) or the Pro*C/C++ Precompiler

Figure 1–8 Generating XML Documents Using XDK C++ Components

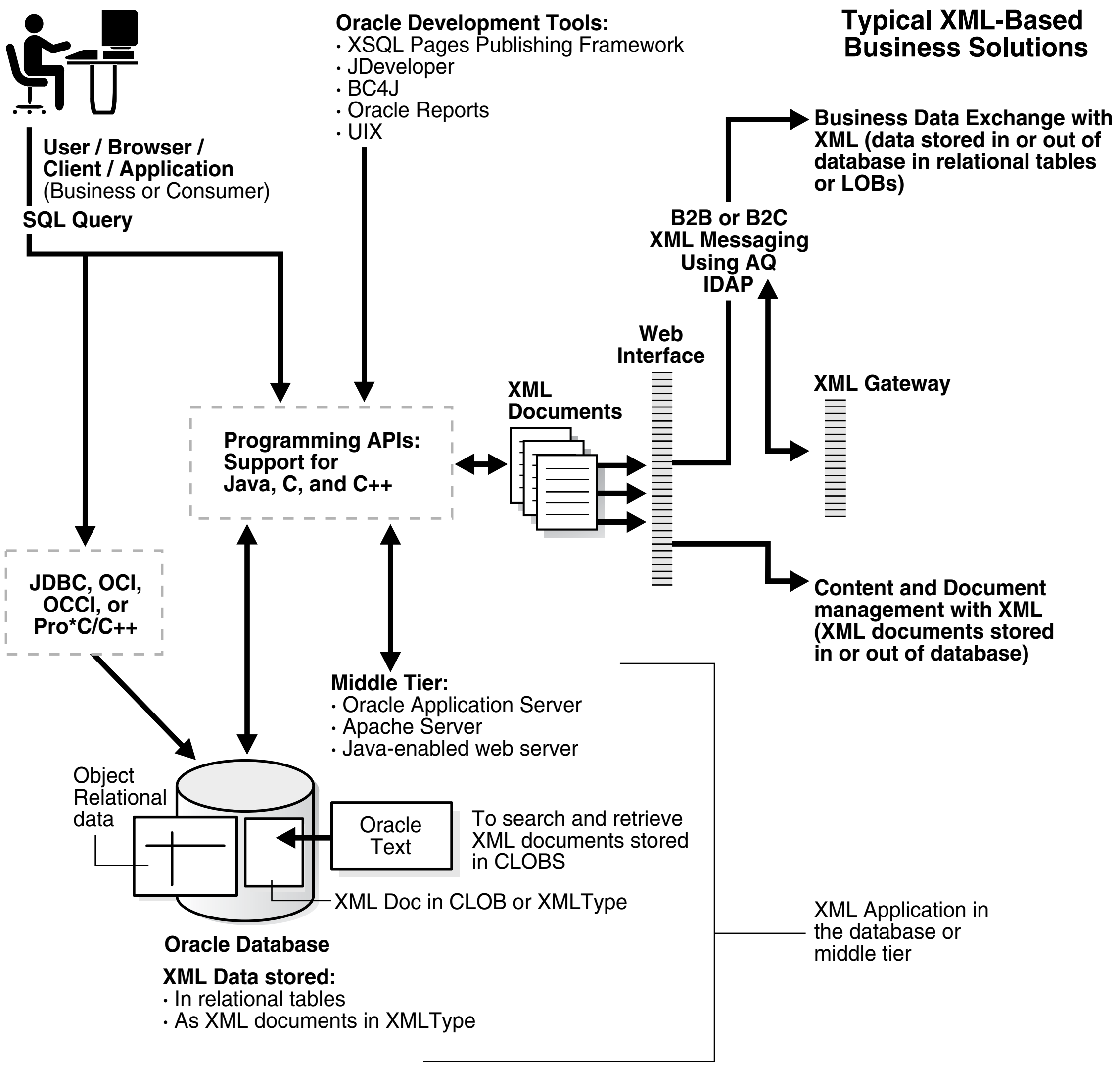

User / Browser / Client Application
Browser or Application
XML
XML Document with or without a DTD or XML Schema
Class Generator
Checks for errors
Creates C++ source files
XSLT Processor
XSLT API is in the XML Parser
XML Document from LOB
Stream
XML Type
DOM or Sax
· Parsed DTD objects
· Parsed HTML
SQL Query
OCCI or Pro*C/C++
XML Parser is within the user application
DTD or XML Schema
Formatted and customized XML Document
XSL Stylesheet
Object Relational data
Oracle Text
LOBs
Oracle database
XML documents stored:
· As single object with tags in CLOB or BLOB
· As data distributed untagged across tables
· Via views that combine the documents and data

XDK and Application Development Tools

Figure 1–9 shows an overview of how the Oracle XML components enable development of E-business solutions.

A user who is a consumer or works for a business, sends SQL queries to an Oracle database either through a Java, C, or C++ application. These applications as well as development tools such as XSQL Pages Publishing Framework, JDeveloper, and so on, transform data from the database into XML documents. These XML documents are input to XML-based business solutions for data exchange with other users, content and data management, and other uses listed in the illustration.

Figure 1–9 Oracle XML Components and E-Business Solutions


Oracle Development Tools:
· XSQL Pages Publishing Framework
· JDeveloper
· BC4J
· Oracle Reports
· UIX
Typical XML-Based Business Solutions
User / Browser / Client / Application (Business or Consumer)
SQL Query
Business Data Exchange with XML (data stored in or out of database in relational tables or LOBs)
B2B or B2C XML Messaging Using AQ IDAP
Web Interface
XML Gateway
XML Documents
Programming APIs: Support for Java, C, and C++
JDBC, OCI, OCCI, or Pro*C/C++
Content and Document management with XML (XML documents stored in or out of database)
Middle Tier:
· Oracle Application Server
· Apache Server
· Java-enabled web server
Object Relational data
Oracle Text
To search and retrieve XML documents stored in CLOBS
XML Doc in CLOB or XMLType
XML Application in the database or middle tier
Oracle Database
XML Data stored:
· In relational tables
· As XML documents in XMLType

The following topics are presented in this section:

- Browsers That Support XML
- Oracle XML Gateway
- JDeveloper
- User Interface XML (UIX)
- Recommended Books and Web Sites About XML

Browsers That Support XML

The following browsers support the display of XML:

- Opera. XML, in version 4.0 and higher
- Citec Doczilla. XML and SGML browser
- Indelv. Displays XML documents only using XSL

- Mozilla Gecko. Supports XML, CSS1, and DOM1
- HP ChaiFarer. Embedded environment that supports XML and CSS1
- ICESoft embedded browser. Supports XML, DOM1, CSS1, and MathML
- Microsoft IE5. Has a full XML parser, IE5.*x* or higher
- Netscape 5.x or higher

Oracle XML Gateway

XML Gateway is a set of services that enables easy 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 is then transmitted to or from the business partner through any message transport agent.

See Also:

- *Oracle Streams Advanced Queuing User's Guide and Reference*
- *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 to Oracle data and features from any .NET application. `ODP.NET` also uses and inherits classes and interfaces available in the Microsoft .NET Framework Class Library.

`ODP.NET` enables the extraction of data from relational and object-relational tables and views as XML documents using the Oracle XDK. 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 Oracle XML Database (Oracle XML DB).

`ODP.NET` supports XML with the following features:

- 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 using Microsoft .NET framework.
- Save changes to the database server using 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*

JDeveloper

Oracle JDeveloper is a J2EE development environment with end-to-end support for developing, debugging, and deploying e-business applications. JDeveloper empowers users with highly productive tools, such as the industry's fastest Java debugger, a new profiler, and the innovative CodeCoach tool for code performance analysis and improvement.

To maximize productivity, JDeveloper provides a comprehensive set of integrated tools that support the complete development life cycle, from source code control, modeling, and coding through debugging, testing, profiling, and deployment. JDeveloper simplifies J2EE development by providing wizards, editors, visual design tools, and deployment tools to create high-quality standard J2EE components, including applets, JavaBeans, Java Server Pages (JSP), servlets, and Enterprise JavaBeans (EJB). JDeveloper also provides a public API to extend and customize the development environment and seamlessly integrate it with external products.

The Oracle XDK is integrated into JDeveloper, offering many ways to create, handle, and transform XML. For example, with the XSQL Servlet, developers can query and manipulate database information, generate XML documents, transform the documents using XSLT stylesheets, and make them available 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, with tag insight to help you easily enter the correct elements and attributes as defined by the schema.

An XML Schema Definition defines the structure of an XML document and is used in the editor to validate the XML and help developers when typing. This feature is called *Code Insight* and provides a list of valid alternatives for XML elements or attributes in the document. Just by specifying the schema for a certain language, the editor can assist you in creating a document in that markup language.

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. These include the following:

- Color-coded syntax highlighting for XML
- Built-in syntax checking for XML and Extensible Style Sheet Language (XSL)
- XSQL Pages and Servlet support, where developers can edit and debug Oracle XSQL Pages, Java programs that can query the database and return formatted XML or insert XML into the database without writing code. The integrated servlet engine enables you to view XML output generated by Java code in the same environment as your program source, making it easy to do rapid, iterative development and testing.
- Includes Oracle's XML Parser for Java
- Includes XSLT Processor
- Related XDK for JavaBeans components
- XSQL Page Wizard
- XSQL Action Handlers
- Schema-driven XML editor

See Also:

- http://www.oracle.com/technology/products/jdev/
- The online discussion forum for JDeveloper is located at http://www.oracle.com/technology/forums

User Interface XML (UIX)

UIX (User Interface XML) is a set of technologies that constitute a framework for building Web applications. The main focus of UIX is the user presentation layer of an application, with additional functionality for managing events and for managing the state of the application flow. UIX is designed to create applications with page-based navigation, such as an online human resources application, rather than full-featured applications requiring advanced interaction, such as an integrated development environment (IDE).

See Also: For sample JDeveloper Demonstration code for UIX:

- http://www.oracle.com/technology/sample_code/products/jdev/content.html
- The complete *UIX Developer's Guide* is included in the JDeveloper online help.

Recommended Books and Web Sites About XML

Here is another XML Frequently Asked Question site:

- http://www.ucc.ie/xml/

Here are some books and Web sites about XML. URLs are often changed, so some URLs in this list are not active links:

- The publisher WROX has a number of helpful books. One of these, *XML Design and Implementation* by Paul Spencer, covers XML, XSL and development.
- *Building Oracle XML Applications* by Steve Muench (published by O'Reilly) See http://www.oreilly.com/catalog/orxmlapp/
- *The XML Bible.* See `http://www.ibiblio.org/xml/books/biblegold/`
- *Oracle9i XML Handbook* by the Oracle XML Product Development Team at http://www.osborne.com/oracle/
- *XML, Java, and the Future of the Web* by Jon Bosak, Sun Microsystems `http://metalab.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm`
- *XML for the Absolute Beginner* by Mark Johnson, JavaWorld `http://www.javaworld.com/jw-04-1999/jw-04-xml_p.html`
- *XML And Databases* by Ronald Bourret, Technical University of Darmstadt `http://www.informatik.tu-darmstadt.de/DVS1/staff/bourret/XML/`
- `XMLAndDatabases.htm` and the 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`

- The XML FAQ by the W3C XML Special Interest Group (the industry clearing house for XML DTDs that allow companies to exchange XML data)
  `http://www.ucc.ie/xml/ XML.org`
- `http://xml.org/`
- xDev (the DataChannel XML Developer pages)
  `http://xdev.datachannel.com/`

Using Oracle XML-Enabled Technology

This section includes general information about Oracle XML-enabled technology, contained in the topics:

- Information for Using the XDK
- Information About Previous Oracle Releases
- XML Standards That Oracle Supports
- Maximum XML File Sizes

Information for Using the XDK

Here are topics about using the XDK:

Using Apache Web Server Instead of the Oracle9*i* Application Server

You can use the Apache Web server which must now interact with Oracle through JDBC or other means. You can use the XSQL servlet. This is a servlet that can run on any servlet-enabled Web server. This runs on Apache and connects to the Oracle database through a **Java Database Connectivity (JDBC)** driver.

Need for an XML Parser If all the XML Was Created By Programs

Whether you still need an XML parser if all XML was created by your programs depends on what you intend to do with the generated XML. If your task is just to generate XML and send it out then you might not need it. But if you wanted to generate an XML DOM tree then you need the Parser. You also need it if you have incoming XML documents and you want to parse and store them. See the XML SQL utility for some help on this issue.

SQL*Loader and Nesting in XML Documents

If you have the following scenario:

```
...
     <something>
        <price>10.00</price>
     </something>
...
   ...
      ...
          <somethingelse>
            <price>55.00</price>
          </somethingelse>
```

Is there a way to uniquely identify the two `<price>` elements?

Answer: No. The field description in the control file can be nested, which is part of the support for object relational columns. The data record to which this maps is, of course, flat but using all the data field description features of the SQL*Loader one can get a lot done. For example:

`sample.xml`

```
<resultset>
    <emp>
        <first>...</first>
        <last>...</last>
        <middle>....</middle>
    </emp>
    <friend>
        <first>...</first>
        <last>...</last>
        <middle>....</middle>
    </friend>
</resultset>
```

`sample.ctl` -- field definition part of the SQL Loader control file

```
field list ....
(
 emp  COLUMN OBJECT ....
   (
        first      char(30)   enclosed by "<first>" and "</first>",
        last       char(30)   enclosed by "<last>" and "</last>",
        middle     char(30)   enclosed by "<middle>" and "</middle>"
   )
    friend COLUMN OBJECT ....
  (
        first      char(30)   enclosed by "<first>" and "</first>",
        last       char(30)   enclosed by "<last>" and "</last>",
        middle     char(30)   enclosed by "<middle>" and "</middle>"
  )
```

Keep in mind that the `COLUMN OBJECT` field names have to match the object column in the database. You will have to use a custom record terminator, otherwise it defaults to `newline` (that is, the `newline` separates data for a complete database record).

If your XML is more complex and you are trying to extract only select fields, you can use `FILLER` fields to reposition the scanning cursor, which scans from where it has left off toward the end of the record (or for the first field, from the beginning of the record).

The SQL*Loader has a very powerful text parser. You can use it for loading XML when the document is very big.

Information About Previous Oracle Releases

These sections concern previous Oracle releases.

Using Oracle Database Version 7 and XML

You can go a long way with Oracle database version 7. The only problem is that you cannot run any of the Java programs inside the server; that is, you cannot load all the XML tools into the server. But you can connect to the database by downloading the Oracle JDBC utility for Oracle database version 7 and run all the programs as client-side utilities.

Doing Data Transfers to Other Vendors Using XML from Oracle Release 7.3.4

Question: My company has Oracle release 7.3.4 and my group is thinking of using XML for some data transfers between us and our vendors. It looks as if we need to move to Oracle8*i* or higher in order to do so. Is there any way of leveraging Oracle release 7 to do XML?

Answer: As long as you have the appropriate JDBC 1.1 drivers for Oracle release 7.3.4 you can use the XML SQL Utility to extract data in XML.

For JDBC drivers, refer to the following Web site for information about Oracle database version 7 JDBC OCI and JDBC Thin Drivers:

`http://www.oracle.com/technology/tech/java/`

Using Versions Prior to Oracle8*i* and Oracle XML Tools?

If I am using an Oracle version earlier than Oracle8*i*, can I supply XML- based applications using Oracle XML tools? If yes, then what are the licensing terms?

The Oracle XDKs for Java, C, and C++ can work outside the database, including the XML SQL Utility and XSQL Pages framework. Licensing is the same, including free runtime. See Oracle Technology Network (OTN) for the latest licenses.

XML Standards That Oracle Supports

Here are discussions about XML standards that Oracle supports.

B2B Standards and Development Tools that Oracle Supports

What B2B XML standards (such as ebXML, cxml, and BizTalk) does Oracle support? What tools does Oracle offer to create B2B exchanges?

Oracle participates in several B2B standards organizations:

- OBI (Open Buying on the Internet)
- ebXML (Electronic Business XML)
- RosettaNet (E-Commerce for Supply Chain in IT Industry)
- OFX (Open Financial Exchange for Electronic Bill Presentment and Payment)

For B2B exchanges, Oracle provides several alternatives depending on customer needs, such as the following:

- Oracle Exchange delivers an out-of-the-box solution for implementing electronic marketplaces
- OracleAS Process Connect has B2B and capability.
- Oracle Gateways for exchanges at data level
- Oracle XML Gateway to transfer XML-based messages from our e-business suite.

The Oracle Internet support provides an integrated and solid platform for B2B exchanges.

Oracle Corporation's Direction Regarding XML

Oracle Corporation's XML strategy is to use XML in ways that exploit all of the benefits of the current Oracle technology stack. Today you can combine Oracle XML components with the Oracle database and Streams to achieve conflict resolution, transaction verification, and so on. Oracle is working to make future releases more seamless for these functions, as well as for functions such as distributed two phase commit transactions.

The `XMLType` datatype is used for storing XML in a column in a table or view.

See Also: *Oracle XML DB Developer's Guide*

XML data is stored either in object-relational tables or views, or as CLOBs. XML transactions are transactions with one of these datatypes and are handled using the standard Oracle mechanisms, including rollback segments, locking, and logging.

From Oracle9*i* onward, Oracle supports sending XML payloads using Streams. This involves making it possible to query XML from SQL.

Oracle is active in all XML standards initiatives, including W3C XML Working Groups, Java Extensions for XML, Open Applications Group, and `XML.org` for developing and registering specific XML schemas.

Oracle Corporation's Plans for XML Query

Oracle is participating in the W3C Working Group for XML Query. Oracle is considering plans to implement a language that enables querying XML data, such as in the XSQL proposal. While XSLT provides static XML transformation features, a query language will add data query flexibility similar to what SQL does for relational data.

Oracle has representatives participating actively in the following W3C Working Groups related to XML and XSL: XML Schema, XML Query, XSL, XLink/**XPointer**, XML Infoset, DOM, and XML Core.

Maximum XML File Sizes

Here are maximum XML file sizes.

Limitations on the Size of an XML File

There are no XML limitations to an XML file size except the limit of the operating system.

Size Limit for XML Documents Generated from the Database

Oracle is not aware of any limits beyond those imposed by the object view and the underlying table structure.

Maximum Size for an XML Document for PL/SQL

Is there a maximum size for an XML document to provide data for PL/SQL (or SQL) across tables, given that no CLOBs are used? The size limit for an XML document providing data for PL/SQL across tables should be what can be inserted into an object view.

2 Getting Started with XDK Java Components

This chapter contains these topics:

XDK Java Components Specifications

XDK Java components, release 10.1, are built on these specifications:

- XML 1.0 (Second Edition)
- DOM Level 2.0 Specifications
  - DOM Level 2.0 Core
  - DOM Level 2.0 Traversal and Range
  - DOM Level 2.0 Events
- DOM Level 3.0 Specifications
  - DOM Level 3.0 Load and Save (internal draft version 10 October 2003)
  - DOM Level 3.0 Validation (Candidate Recommendation 30 July 2003)
- SAX 2.0 and SAX Extensions
- XSLT/XPath 2.0 Specifications
  - XSL Transformations (XSLT) 2.0 (working draft dated 02 May 2003)
  - XML Path Language (XPath) 2.0 (working draft dated 22 August 2003)
  - XPath 2.0 Data Model (working draft dated 11th November 2002)
- XML Schema Specifications
  - XML Schema Part 0: Primer
  - XML Schema Part 1: Structures
  - XML Schema Part 2: Datatypes

- XML Pipeline Definition Language 1.0
- Java API for XML Processing 1.1 and 1.2 (JAXP)
- Java Architecture for XML Binding 1.0 (JAXB)

DOM Specifications

In release 10.1, the DOM APIs include support for two new working drafts, DOM Level 3 Validation and DOM Level 3 Load and Save.

Load and Save

The DOM Level 3 Load and Save module enables software developers to load and save XML content inside conforming products. 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 following configuration parameters are supported by `XMLDOMBuilder` which implements `LSParser`:

- "cdata-sections"
- "validate"
- "validate-if-schema"
- "whitespace-in-element-content"

The following configuration parameters are supported by `XMLDOMWriter` which implements `LSSerializer`:

- "format-pretty-print"
- "xml-declaration"

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:

`NameList` and `DOMStringList` in DOM core are supported for validation purpose.

Validation is based on XML Schema, DTD needs to be converted to Schema first (use `DTDToSchema` utility).

XSL Transformations Specifications

The XSLT processor adds support for the current working drafts of XSLT 2.0, XPath 2.0, and the shared XPath/XQuery data model.

For the XPath 2.0 specification, only the new XPath 2.0 grammar and backwards compatibility with XPath 1.0 are supported.

These features of the specifications are not supported in release 10.1:

- The functions in the Functions and Operators specification are not supported. Only the functions from XSLT 1.0 specification are supported.
- The `validate` and `complex` types in `SequenceType` expressions are not supported.

- The new datatypes `fn:yearMonthduration` and `fn:dayTimeDuration` are not supported.
- The Schema Import and Static Typing features are not supported.
- The XSLT instructions `xsl:result-document` and `xsl:namespace` are not supported.
- The XSLT instructions `xsl:text` and `xsl:number` use XSLT 1.0 semantics and syntax.
- The standard attributes are allowed only on `xsl:stylesheet` and literal result elements, except for `default-xpath-namespace` and `version`.
- The processor does not honor the following attributes:
  - `[required]` on `xsl:param`
  - [XML Schema related attributes, like `xsl:validation` and `xsl:type`, etc.
- Regular expression functions are not supported.
- Parameters are not passed through built-in templates.
- `xsl:sequence` is not supported

Installing XDK Java Components

XDK Java components are included with the Oracle database and with the Oracle application server. You can download the latest beta or production version of XDK Java components from OTN as part of the XDK. The XDK Java components and JavaBeans are now bundled together.

If you installed XDK with the Oracle database or the Oracle application server, you can use this chapter as a reference.

If you download the XDK from OTN, follow these steps:

- Go to the URL:

  `http://www.oracle.com/technology/tech/xml/xdk/content.html`

- Click the Software link on the right-side of the page.
- Logon with your OTN username and password (registration is free if you do not already have an account).
- Select the Windows or UNIX download.
- Select the appropriate download for your operating system.
- Accept all terms of the licensing agreement and then download the software by clicking the appropriate distribution.
- Extract the files in the distribution:
  - Choose a directory under which you want the `./xdk` directory and subdirectories to go.
  - Change to that directory and then extract the XDK Java components download archive file. For UNIX:

    ```
    tar xvfz xdk_XXXX.tar.gz       # UNIX. XXXX is the release name
    Use WinZip visual archive extraction tool in Windows
    ```

XDK Java Components Directory Structure

After installing the XDK, the directory structure is:

```
-$XDK_HOME
    | - bin: executable files and setup script or batch files.
    | - lib: library files.
    | - xdk:
         | - admin: (Administration): SQL script and XSL Servlet Configuration
                   file (XSQLConfig.xml).
         | - demo/java: demonstration code
         | - doc/java: documents including release notes and Javadoc HTML.
```

All the XDK Java components are certified and supported with JDK 1.2, JDK 1.3, and JDK 1.4. Make sure that your **CLASSPATH** includes all the necessary libraries:

Table 2–1 XDK Java Components Libraries

Component	Library	Notes
XML Parser, XSL Processor	`xmlparserv2.jar`	XML Parser V2 for Java, which includes JAXP 1.1, DOM, SAX and XSLT APIs.
Message files for XML Parser.	`xmlmesg.jar`	If you want to use XML Parser with a language other than English, you need to set this JAR file in your `CLASSPATH`.
XML Schema Processor	`xschema.jar`	XML Schema Processor for Java.
XML SQL Utility	`xsu12.jar`	XML SQL Utility for JDK 1.2 and later.
XSQL Servlet	`oraclesql.jar`	Oracle XSQL Servlet.
XSQL	`xsqlserializers.jar`	Oracle XSQL Serializers for FOP/PDF Integration.
JAXB Class Generator, Pipeline Processor, Differ	`xml.jar`	Class Generator for Java.
JavaBeans	`xmlcomp.jar` `xmlcomp2.jar`	JavaBeans Utilities.
TransX Utility	`transx.zip`	Oracle TransX Utility.

In addition, XML SQL Utility, XSQL Servlet, and TransX Utility all depend on JDBC and globalization support libraries, which are listed in Table 2–2:

Table 2–2 JDBC and Globalization Support Libraries for XDK Java Components

Component	Library	Notes
JDBC	`classes12.zip`	JDBC for JDK 1.2 and later.
Globalization Support	`orai18n.jar`	Globalization support for JDK 1.2 and later.
XMLType	`xdb.jar`	XMLType Java APIs in `$ORACLE_HOME/rdbms/jlib/`
JDeveloper Runtime	`jdev-rt.zip`	Java GUI libraries.

XDK Java Components Environment Settings

The UNIX and Windows environment settings are listed:

UNIX Environment Settings for XDK Java Components

This file sets up the environment:

`$XDK_HOME/bin/env.csh`

Table 2–3 lists the UNIX environment variables, with the ones that must be customized each marked with "Yes":

Table 2–3 UNIX Environment Settings for XDK Java Components

Variable	Notes	Yes/No
`$JDBCVER`	JDBC version. For JDK 1.2 and later, set to 12.	Yes
`$JDKVER`	JDK version obtained by `JDK -version`. Default value is 1.2.2_07.	Yes
`$INSTALL_ROOT`	Installation root of XDK which is the directory `$XDK_HOME`.	No
`$JAVA_HOME`	Directory where the Java JDK, Standard Edition is installed.	Yes

`$CLASSPATHJ`	`{ORACLE_HOME}/jdbc/lib/classes${JDBCVER}.zip:`
	`${ORACLE_HOME}/jdbc/lib/nls_charset${JDBCVER}.jar`
	If you are running the XSU on a system different from where the Oracle database is installed, you have to update your `CLASSPATHJ` setting with the correct locations of the JDBC library `(classes12.jar)`. The `orai18n.jar` is needed to support certain character sets. See "XDK Java Components Globalization Support" on page 2-6. Note that if you do not have these libraries on your system, these are both available on OTN (`http://www.oracle.com/technology`), as part of the JDBC driver download.
		Yes

`$CLASSPATH`	Include the following:
	`.:${CLASSPATHJ}:${INSTALL_ROOT}/lib/xmlparserv2.jar:`
	`${INSTALL_ROOT}/lib/xschema.jar:`
	`${INSTALL_ROOT}/lib/xsu${JDBCVER}.jar:`
	`${INSTALL_ROOT}/lib/oraclexsql.jar:`
	`${INSTALL_ROOT}/lib/classgen.jar`
		No

`$PATH`	`${JAVA_HOME}/bin:${PATH}:${INSTALL_ROOT}/bin`	No
`$LD_LIBRARY_PATH`	For OCI JDBC connections: `${ORACLE_HOME}/lib:${LD_LIBRARY_PATH}`	No

Windows Environment Settings for XDK Java Components

This file sets up the environment:

`%XDK_HOME%\bin\env.bat`

Table 2–4 lists the Windows environment variables with the ones that must be customized each marked with "Yes":

Table 2–4 Windows Environment Settings for XDK Java Components

Variable	Notes	Yes/No
`%JDBCVER%`	JDBC version. If using JDK 1.2 and later, it should be set to 12.	Yes
`%JDKVER%`	JDK version which you can get from: `JDK -version`. Default value is 1.2.2_07.	Yes
`%INSTALL_ROOT%`	Installation root of XDK, which is the directory `%XDK_HOME%`.	No
`%JAVA_HOME%`	Directory where the Java SDK, Standard Edition is installed.	Yes
`%CLASSPATHJ%`	`CLASSPATHJ=%ORACLE_HOME%\jdbc\lib\classes%JDBCVER%.zip;` `%ORACLE_HOME%\jdbc\lib\nls_charset%JDBCVER%.jar`	Yes
`%CLASSPATH%`	`.;%CLASSPATHJ%;%INSTALL_ROOT%\lib\xmlparserv2.jar;` `%INSTALL_ROOT%\lib\xschema.jar;` `%INSTALL_ROOT%\lib\xsu%JDBCVER%.jar;` `%INSTALL_ROOT%\lib\oraclexsql.jar;%INSTALL_ROOT%\lib\classgen.jar`	No
`%PATH%`	`PATH=%JAVA_HOME%\bin;%ORACLE_HOME%\bin;%PATH%;%INSTALL_ROOT%\bin`	No

XDK Java Components Globalization Support

Here is a summary on the settings that relate to Globalization Support:

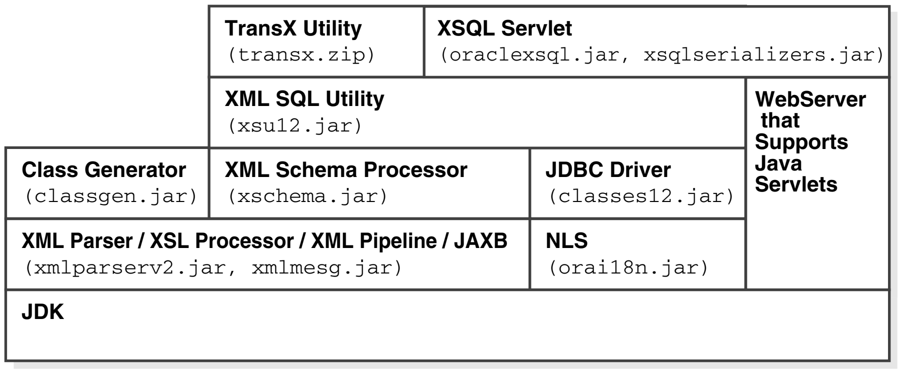
- Using `xmlmesg.jar`: If you are using a language other than English you need to set the `xmlmesg.jar` into your `CLASSPATH` to let the parser get correct messages in your language.
- Using `orai18n.jar`: If you are using a multibyte character set other than one of the following,
  - UTF-8
  - ISO8859-1
  - JA16SJIS

  then you must set this JAR file into your Java `CLASSPATH` so that JDBC can convert the character set of the input file to the database character set during the loading of XML files using either XSU, TransX or XSQL Servlet.

XDK Java Components Dependencies

Figure 2–1 shows the dependencies of XDK Java Components when using JDK 1.2 and higher:

Figure 2–1 XDK Java Components Dependencies Using JDK 1.2.x and Higher



After you correctly setup the environment, include all the necessary JAR files in your CLASSPATH. You can then start writing your Java programs and compiling them with the `javac` command:

```
javac your_program.java
```

If the compilation finishes without errors, then you can just test your program using the command line or the Web Server.

> **See Also:** Chapter 3, "XML Parser for Java" for further discussion of the XDK Java components

Verifying the XDK Java Components Version

To obtain the version of XDK you are working with, compile and run the following Java code (`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());
   }

}
```

3 XML Parser for Java

This chapter contains these topics:

XML Parser for Java Overview

Oracle provides XML parsers for Java, C, C++, and PL/SQL. This chapter discusses the parser for Java only. Each of these parsers is a standalone XML component that parses an XML document (and possibly also a standalone document type definition (DTD) or XML Schema) so that they can be processed by your application. In this chapter, the application examples presented are written in Java.

XML Schema is a W3C XML recommendation effort to introduce the concept of data types to XML documents and replace the syntax of DTDs with one which is based on XML. The process of checking the syntax of XML documents against a DTD or XML Schema is called validation.

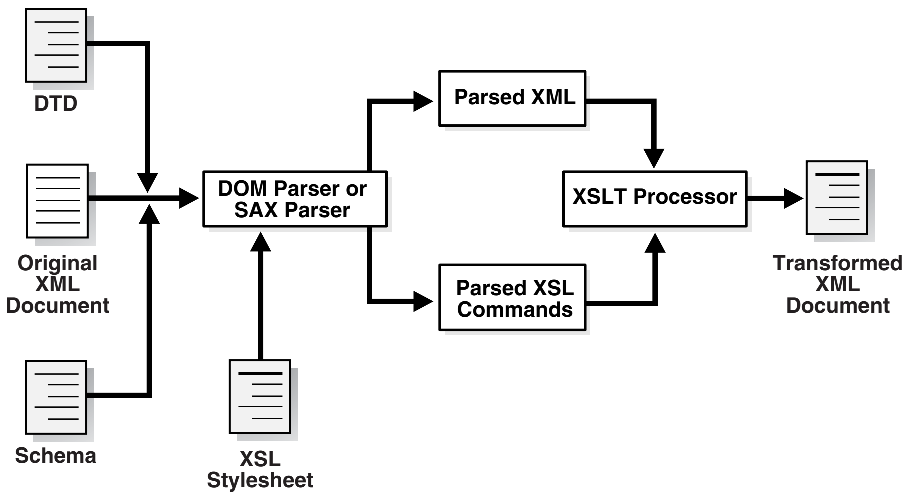
To use an external DTD, include a reference to the DTD in your XML document. Without it there is no way for the parser to know what to validate against. Including the reference is the XML standard way of specifying an external DTD. Otherwise you need to embed the DTD in your XML Document.

Figure 3–1 shows an XML document as input to the XML Parser for Java. The DOM or SAX parser interface parses the XML document. The parsed XML is then transferred to the application for further processing.

The XML Parser for Java includes an integrated XSL Transformation (XSLT) Processor for transforming XML data using XSL stylesheets. Using the XSLT Processor, you can transform XML documents from XML to XML, XML to HTML, or to virtually any other text-based format.

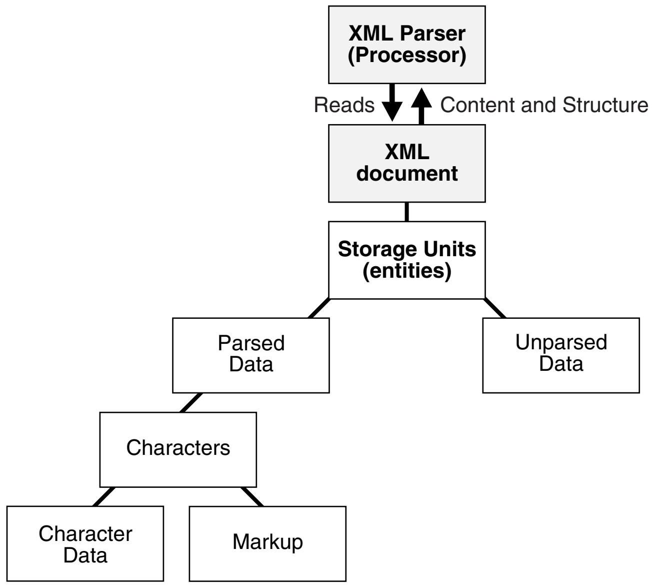
If a stylesheet is used, the DOM or SAX interface also parses and outputs the XSL commands. These are sent together with the parsed XML to the XSLT Processor where the selected stylesheet is applied and the transformed (new) XML document is then output. Figure 3–1 shows a simplified view of the XML Parser for Java.

Figure 3–1 XML Parser for Java



The XML Parser for Java processor reads XML documents and provides access to their content and structure. An XML processor does its work on behalf of another module, your application. This parsing process is illustrated in Figure 3–2.

Figure 3–2 XML Parsing Process



See Also:

- Schema Primer at http://www.w3.org/TR/xmlschema-0/
- Schema structures at http://www.w3.org/TR/xmlschema-1/
- Schema datatypes at http://www.w3.org/TR/xmlschema-2/
- Chapter 4, "XSLT Processor for Java"
- **Oracle XML DB Developer's Guide** for a discussion of the PL/SQL Parser
- **Oracle XML API Reference** for methods of the XML Parser for Java

Namespace Support

The XML Parser for Java also supports XML Namespaces. Namespaces are a mechanism to resolve or avoid name collisions between element types (tags) or attributes in XML documents.

This mechanism provides "universal" namespace element types and attribute names. Such tags are qualified by uniform resource identifiers (URIs), such as:

```
<oracle:EMP xmlns:oracle="http://www.oracle.com/xml"/>
```

For example, namespaces can be used to identify an Oracle `<EMP>` data element as distinct from another company's definition of an `<EMP>` data element. This enables an application to more easily identify elements and attributes it is designed to process.

The XML Parser for Java can parse universal element types and attribute names, as well as unqualified "local" element types and attribute names.

See Also:

- Chapter 5, "XML Schema Processor for Java"
- **Oracle XML API Reference** for methods of the XML Parser for Java
- http://www.w3.org/TR/1999/REC-xml-names-19990114/ for the W3C Recommendation for XML Namespaces

XML Parser for Java Validation Modes

Validation involves checking whether or not the attribute names and element tags are legal, whether nested elements belong where they are, and so on.

The DTD file defined in the `<!DOCTYPE>` declaration must be relative to the location of the input XML document. Otherwise, you need to use the `setBaseURL(url)` functions to set the base URL to resolve the relative address of the DTD if the input is coming from `InputStream`.

If you are parsing an `InputStream`, the parser does not know where that `InputStream` came from, so it cannot find the DTD in the same directory as the current file. The solution is to `setBaseURL()` on `DOMParser()` to give the parser the URL hint information to be able to derive the rest when it goes to get the DTD.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup.

Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

The parser method `setValidationMode(mode)` parses XML in the mode values shown in Table 3–1.

Table 3–1 XML Parser for Java Validation Modes

Name of Mode	Mode Value in Java	Description
Non-Validating Mode	`NONVALIDATING`	The parser verifies that the XML is well-formed and parses the data into a tree of objects that can be manipulated by the DOM API.
DTD Validating Mode	`DTD_VALIDATION`	The parser verifies that the XML is well-formed and validates the XML data against the DTD (if any).
Partial Validation Mode	`PARTIAL_VALIDATION`	Partial validation validates all or part of the input XML document according to the DTD or XML Schema, if one is present. If one is not present, the mode is set to Non-Validating Mode. With this mode, the schema validator locates and builds schemas and validates the whole or a part of the instance document based on the schemaLocation and noNamespaceSchemaLocation attributes. See code example`XSDSample.java` in directory `/xdk/demo/java/schema`.
Schema Validation Mode	`SCHEMA_VALIDATION`	The XML Document is validated according to the XML Schema specified for the document.
Lax Validation	`SCHEMA_LAX_VALIDATION`	The validator 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. This is shown in the sample `XSDLax.java` in the schema directory.
Strict Validation	`SCHEMA_STRICT_VALIDATION`	The validator 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.
Auto Validation Mode	See description.	If a DTD is available, the mode value is set to `DTD_VALIDATION`, if a Schema is present then it is set to `SCHEMA_VALIDATION`. If neither is available, it is set to `NONVALIDATING` mode value, which is the default.

In addition to the validator to build the schema itself, you can use `XSDBuilder` to build schemas and set it to the validator using `setXMLSchema() method`. See code example `XSDSetSchema.java`. By using the `setXMLSchema()` method, the validation mode is automatically set to `SCHEMA_STRICT_VALIDATION`, and both `schemaLocation` and `noNamespaceSchemaLocation` attributes are ignored. You can also change the validation mode to `SCHEMA_LAX_VALIDATION`.

Using DTDs with the XML Parser for Java

The following is a discussion of the use of DTDs. It contains the sections:

- Enabling DTD Caching
- Recognizing External DTDs
- Loading External DTDs from a JAR File
- Checking the Correctness of Constructed XML Documents
- Parsing a DTD Object Separately from an XML Document
- XML Parsers Case-Sensitivity
- Allowed File Extensions in External Entities
- Creating a DOCUMENT_TYPE_NODE
- Standard DTDs That Can be Used for Orders, Shipments, and So On

Enabling DTD Caching

DTD caching is optional and is not enabled automatically.

The XML Parser for Java provides for validating and non-validating DTD caching through the `setDoctype()` function. After you set the DTD using this function, XMLParser will cache this DTD for further parsing.

If your application has to parse several XML documents with the same DTD, after you parse the first XML document, you can get the DTD from parser and set it back:

```
dtd = parser.getDoctype();
parser.setDoctype(dtd);
```

The parser will cache this DTD and use it for parsing the following XML documents.

Set the `DOMParser.USE_DTD_ONLY_FOR_VALIDATION` attribute, if the cached DTD Object is used only for validation by:

```
parser.setAttribute(DOMParser.USE_DTD_ONLY_FOR_VALIDATION,Boolean.TRUE);
```

Otherwise, the XML parser will copy the DTD object and add it to the result DOM tree.

The method to set the DTD is `setDoctype()`. Here is an example:

```
// Test using InputSource
parser = new DOMParser();
parser.setErrorStream(System.out);
parser.showWarnings(true);

FileReader r = new FileReader(args[0]);
InputSource inSource = new InputSource(r);
inSource.setSystemId(createURL(args[0]).toString());
parser.parseDTD(inSource, args[1]);
dtd = (DTD)parser.getDoctype();

r = new FileReader(args[2]);
```

```
inSource = new InputSource(r);
inSource.setSystemId(createURL(args[2]).toString());
// ********************
parser.setDoctype(dtd);
// ********************
parser.setValidationMode(DTD_VALIDATION);
parser.parse(inSource);

doc = (XMLDocument)parser.getDocument();
doc.print(new PrintWriter(System.out));
```

Recognizing External DTDs

To recognize external DTDs, the XML Parser for Java has the `setBaseURL()` method.

The way to redirect the DTD is by using `resolveEntity()`:

1. Parse your External DTD using a DOM parser's `parseDTD()` method.
2. Call `getDoctype()` to get an instance of `oracle.xml.parser.v2.DTD`.
3. On the document where you want to set your DTD programmatically, use the call `setDoctype(yourDTD)`. Use this technique to read a DTD out of your product's `JAR` file.

Loading External DTDs from a JAR File

The parser supports a base URL method (`setBaseURL()`), but that just points to a place where all the DTDs are exposed.

Do the following steps:

1. Load the DTD as an `InputStream`:

   ```
   InputStream is =      YourClass.class.getResourceAsStream("/foo/bar/your.dtd");
   ```

   This opens `./foo/bar/your.dtd` in the first relative location on the `CLASSPATH` that it can be found, including out of your JAR if it is in the `CLASSPATH`.
2. Parse the DTD:

   ```
   DOMParser d = new DOMParser();
   d.parseDTD(is, "rootelementname");
   d.setDoctype(d.getDoctype());
   ```

3. Parse your document:

   ```
   d.parse("yourdoc");
   ```

Checking the Correctness of Constructed XML Documents

No validation is done while creating the DOM tree using DOM APIs. So setting the DTD in the document does not help validate the DOM tree that is constructed. The only way to validate an XML file is to parse the XML document using the DOM parser or the SAX parser.

Parsing a DTD Object Separately from an XML Document

The `parseDTD()` method enables you to parse a DTD file separately and get a DTD object. Here is some sample code to do this:

```
DOMParser domparser = new DOMParser();
domparser.setValidationMode(DTD_VALIDATION);
/* parse the DTD file */
```

```
domparser.parseDTD(new FileReader(dtdfile));
DTD dtd = domparser.getDoctype();
```

XML Parsers Case-Sensitivity

XML is inherently case-sensitive, therefore the parsers enforce case sensitivity in order to be compliant. When you run in non-validation mode only well-formedness counts. However `<test></Test>` signals an error even in non-validation mode.

Allowed File Extensions in External Entities

The file extension for external entities is unimportant so you can change it to any convenient extension, including no extension.

Creating a DOCUMENT_TYPE_NODE

There is no way to create a new `DOCUMENT_TYPE_NODE` object using the DOM APIs. The only way to get a DTD object is to parse the DTD file or the XML file using the DOM parser, and then use the `getDocType()` method.

The following statement does not create a DTD object. It creates an `XMLNode` object with the type set to `DOCUMENT_TYPE_NODE`, which in fact is not allowed. The `ClassCastException` is raised because `appendChild` expects a DTD object (based on the type).

```
appendChild(New XMLNode("test",Node.DOCUMENT_TYPE_NODE));
```

Standard DTDs That Can be Used for Orders, Shipments, and So On

Basic, standard DTDs to build on for orders, shipments, and acknowledgements are found on this Web site, which has been set up for that purpose:

`http://www.xml.org/`

About DOM and SAX APIs

XML APIs for parsing are of two kinds:

- DOM APIs (Tree-based)
- SAX APIs (Event-based)

Consider the following simple XML document:

```
<?xml version="1.0"?>
  <EMPLIST>
    <EMP>
     <ENAME>MARY</ENAME>
    </EMP>
    <EMP>
     <ENAME>SCOTT</ENAME>
    </EMP>
  </EMPLIST>
```

DOM: Tree-Based API

A tree-based API (such as DOM) builds an in-memory tree representation of the XML document. It provides classes and methods for an application to navigate and process the tree.

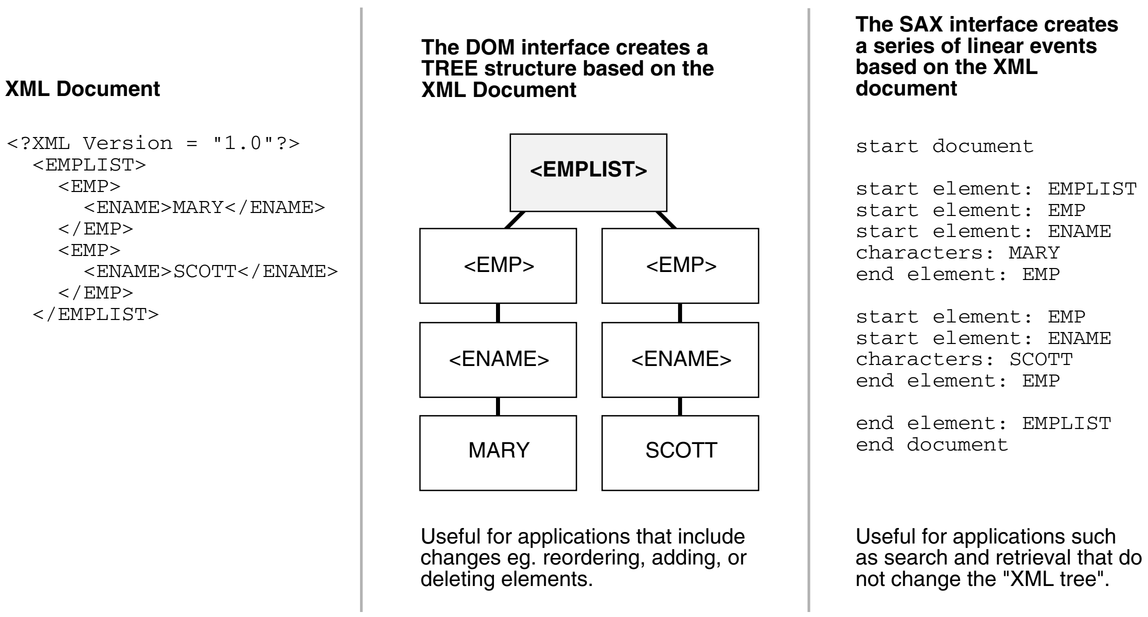
In general, the DOM interface is most useful for structural manipulations of the XML tree, such as reordering elements, adding or deleting elements and attributes, renaming elements, and so on. For example, for the immediately preceding XML document, the DOM creates an in-memory tree structure as shown inFigure 3–3.

SAX: Event-Based API

An event-based API (such as SAX) uses calls to report parsing events to the application. Your Java application deals with these events through customized event handlers. Events include the start and end of elements and characters.

Unlike tree-based APIs, event-based APIs usually do not build in-memory tree representations of the XML documents. Therefore, in general, SAX is useful for applications that do not need to manipulate the XML tree, such as search operations, among others. The preceding XML document becomes a series of linear events as shown in Figure 3–3.

Figure 3–3 Comparing DOM (Tree-Based) and SAX (Event-Based) APIs


XML Document
The DOM interface creates a TREE structure based on the XML Document
<EMPLIST>
<EMP>
<EMP>
<ENAME>
<ENAME>
MARY
SCOTT
Useful for applications that include changes eg. reordering, adding, or deleting elements.
The SAX interface creates a series of linear events based on the XML document
Useful for applications such as search and retrieval that do not change the "XML tree".

```
<?XML Version = "1.0"?>
  <EMPLIST>
    <EMP>
      <ENAME>MARY</ENAME>
    </EMP>
    <EMP>
      <ENAME>SCOTT</ENAME>
    </EMP>
  </EMPLIST>
```

```
start document

start element: EMPLIST
start element: EMP
start element: ENAME
characters: MARY
end element: EMP

start element: EMP
start element: ENAME
characters: SCOTT
end element: EMP

end element: EMPLIST
end document
```

Guidelines for Using DOM and SAX APIs

Here are some guidelines for using the DOM and SAX APIs:

DOM

- Use the DOM API when you need to use random access.
- Use DOM when you are performing XSL Transformations.
- Use DOM when you are calling XPath. SAX does not support it.
- Use DOM when you want to have tree iterations and need to walk through the entire document tree.
- Customize DOM tree building: `org.w3c.dom.Is.DOMBuilderFilter`.
- Avoid parsing external DTDs if no validation is required: `DOMParser.set.Attribute(DOMParsser.STANDALONE, Boolean.TRUE);`.

- Avoid including the DTD object in DOM unless necessary: `DOMParser.setAttribute(DOMParser.USE_DTD_ONLY_FOR_VALIDATION, Boolean.TRUE);`.
- Use DTD caching for DTD validations: `DOMParser.setDoctype(dtd);`.
- Build DOM asynchronously using DOM 3.0 Load and Save: `DOMImplementationLS.MODE_ASYNCHRONOUS`.
- A unified DOM API supports both `XMLType` columns and XML documents.
- When using the DOM interface, use more attributes than elements in your XML to reduce the pipe size.

> **See Also:** "DOM Specifications" on page 2-2 for information on what is supported for this release

SAX

- Use the SAX API when your data is mostly streaming data.
- Use SAX to save memory. DOM consumes more memory.
- To increase the speed of retrieval of XML documents from a database, use the SAX interface instead of DOM. Make sure to select the `COUNT(*)` of an indexed column (the more selective the index the better). This way the optimizer can satisfy the count query with a few inputs and outputs of the index blocks instead of a full-table scan.
- Use SAX 2.0, because SAX 1.0 is deprecated.
- There are output options for SAX: print formats, XML declaration, CDATA, DTD.
- Multi-task the SAX processing to improve throughput (using multi-handlers and enabling multiple processing in callbacks). Multiple handler registrations per SAX parsing: oracle.xml.parser.V2.XMLMultiHandler.
- Use the built-in XML serializer to simplify output creation: `oracle.xml.parser.V2.XMLSAXSerializer`.

About XML Compressor

The XML Compressor supports binary compression of XML documents. The compression is based on tokenizing the XML tags. The assumption is that any XML document has a repeated number of tags and so tokenizing these tags gives a considerable amount of compression. Therefore the compression achieved depends on the type of input document; the larger the tags and the lesser the text content, then the better the compression.

The goal of compression is to reduce the size of the XML document without losing the structural and hierarchical information of the DOM tree. The compressed stream contains all the "useful" information to create the DOM tree back from the binary format. The compressed stream can also be generated from the SAX events.

XML Parser for Java can also compress XML documents. Using the compression feature, an in-memory DOM tree or the SAX events generated from an XML document are compressed to generate a binary compressed output. The compressed stream generated from DOM and SAX are compatible, that is, the compressed stream generated from SAX can be used to generate the DOM tree and vice versa.

As with XML documents in general, you can store the *compressed* XML data output as a BLOB (Binary Large Object) in the database.

Sample programs to illustrate the compression feature are described in Table 3–2, " XML Parser for Java Sample Programs".

XML Serialization and Compression

An XML document is compressed into a binary stream by means of the serialization of an in-memory DOM tree. When a large XML document is parsed and a DOM tree is created in memory corresponding to it, it may be difficult to satisfy memory requirements and this can affect performance. The XML document is compressed into a byte stream and stored in an in-memory DOM tree. This can be expanded at a later time into a DOM tree without performing validation on the XML data stored in the compressed stream.

The compressed stream can be treated as a serialized stream, but the information in the stream is more controlled and managed, compared to the compression implemented by Java's default serialization.

There are two kinds of XML compressed streams:

- DOM based compression: The in-memory DOM tree, corresponding to a parsed XML document, is serialized, and a compressed XML output stream is generated. This serialized stream regenerates the DOM tree when read back.
- SAX based compression: The compressed stream is generated when an XML file is parsed using a SAX parser. SAX events generated by the SAX parser are handled by the SAX compression utility, which handles the SAX events to generate a compressed binary stream. When the binary stream is read back, the SAX events are generated.

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.

Compression is supported only in the XDK Java components.

Running the Sample Applications for XML Parser for Java

The directory `demo/java/parser` contains some sample XML applications to show how to use the XML Parser for Java. The following are the sample Java files in its subdirectories (`common`, `comp`, `dom`, `jaxp`, `sax`, `xslt`):

Table 3–2 XML Parser for Java Sample Programs

Sample Program	Purpose
`XSLSample`	A sample application using XSL APIs
`DOMSample`	A sample application using DOM APIs
`DOMNamespace`	A sample application using Namespace extensions to DOM APIs
`DOM2Namespace`	A sample application using DOM Level 2.0 APIs
`DOMRangeSample`	A sample application using DOM Range APIs
`EventSample`	A sample application using DOM Event APIs
`NodeIteratorSample`	A sample application using DOM Iterator APIs
`TreeWalkerSample`	A sample application using DOM TreeWalker APIs
`SAXSample`	A sample application using SAX APIs

Table 3–2 (Cont.) XML Parser for Java Sample Programs

Sample Program	Purpose
`SAXNamespace`	A sample application using Namespace extensions to SAX APIs
`SAX2Namespace`	A sample application using SAX 2.0
`Tokenizer`	A sample application using `XMLToken` interface APIs
`DOMCompression`	A sample application to compress a DOM tree
`DOMDeCompression`	A sample to read back a DOM from a compressed stream
`SAXCompression`	A sample application to compress the SAX output from a SAX Parser
`SAXDeCompression`	A sample application to regenerate the SAX events from the compressed stream
`JAXPExamples`	Samples using the JAXP 1.1 API

The `Tokenizer` application implements `XMLToken` interface, which you must register using the `setTokenHandler()` method. A request for the XML tokens is registered using the `setToken()` method. During tokenizing, the parser does not validate the document and does not include or read internal or external utilities.

To run the sample programs:

1. Use `make` (for UNIX) or `make.bat` (for Windows) in the directory `xdk/demo/java` to generate `.class` files.
2. Add `xmlparserv2.jar` and the current directory to the `CLASSPATH`.

The following list does not have to be done in order, except for decompressing:

- Run the sample programs for the DOM APIs and SAX APIs in each directory:

  ```
  java classname sample_xml_file
  ```

- Run the sample program for XSL APIs in its directory:

  ```
  java XSLSample sample_xsl_file sample_xml_file
  ```

- Run the sample program for `Tokenizer` APIs in its directory:

  ```
  java Tokenizer sample_xml_file token_string
  ```

- Run the sample program for compressing a DOM tree in its directory:

  ```
  java DOMCompression sample.dat
  ```

  The compressed output is generated in a file called `xml.ser`.

- Run the sample program to build the DOM tree from the compressed stream if you have done the last step.

  ```
  java DOMDeCompression xml.ser
  ```

- Run the sample program for compressing the SAX events in its directory

  ```
  java SAXCompression sample.dat
  ```

- Run the sample program for regenerating the SAX events from the compressed stream if you have done the last step:

  ```
  java SAXDeCompression xml.ser
  ```

- Run the sample program for the JAXP 1.1 API in its directory:

  ```
  java JAXPExamples
  ```

  The XML document file and stylesheets are given inside the program `JAXPExamples.java`. The `Content Handler` is inside the Java file `oraContentHandler.java`.

Using XML Parser for Java: DOMParser Class

To write DOM-based parser applications you can use the following classes:

- `DOMNamespace` class
- `DOMParser` class
- `XMLParser` class

Since `DOMParser` extends `XMLParser`, all methods of `XMLParser` are also available to `DOMParser`. Figure 3–4, "XML Parser for Java: DOMParser()" shows the main steps you need when coding with the `DOMParser` class.

Without DTD Input

In some applications, it is not necessary to validate the XML document. In this case, a DTD is not required.

1. A new `DOMParser()` is called. Some of the methods to use with this object are:
   - `setValidateMode()`
   - `setPreserveWhiteSpace()`
   - `setDoctype()`
   - `setBaseURL()`
   - `showWarnings()`
2. The results of `DOMParser()` are passed to `XMLParser.parse()` along with the XML input. The XML input can be a file, a string buffer, or URL.
3. Use the `XMLParser.getDocument()` method.
4. Optionally, you can apply other DOM methods such as:
   - `print()`
   - `DOMNamespace()` methods
5. The Parser outputs the DOM tree XML (parsed) document.
6. Optionally, use `DOMParser.reset()` to clean up any internal data structures, once the DOM API has finished building the DOM tree.

With a DTD Input

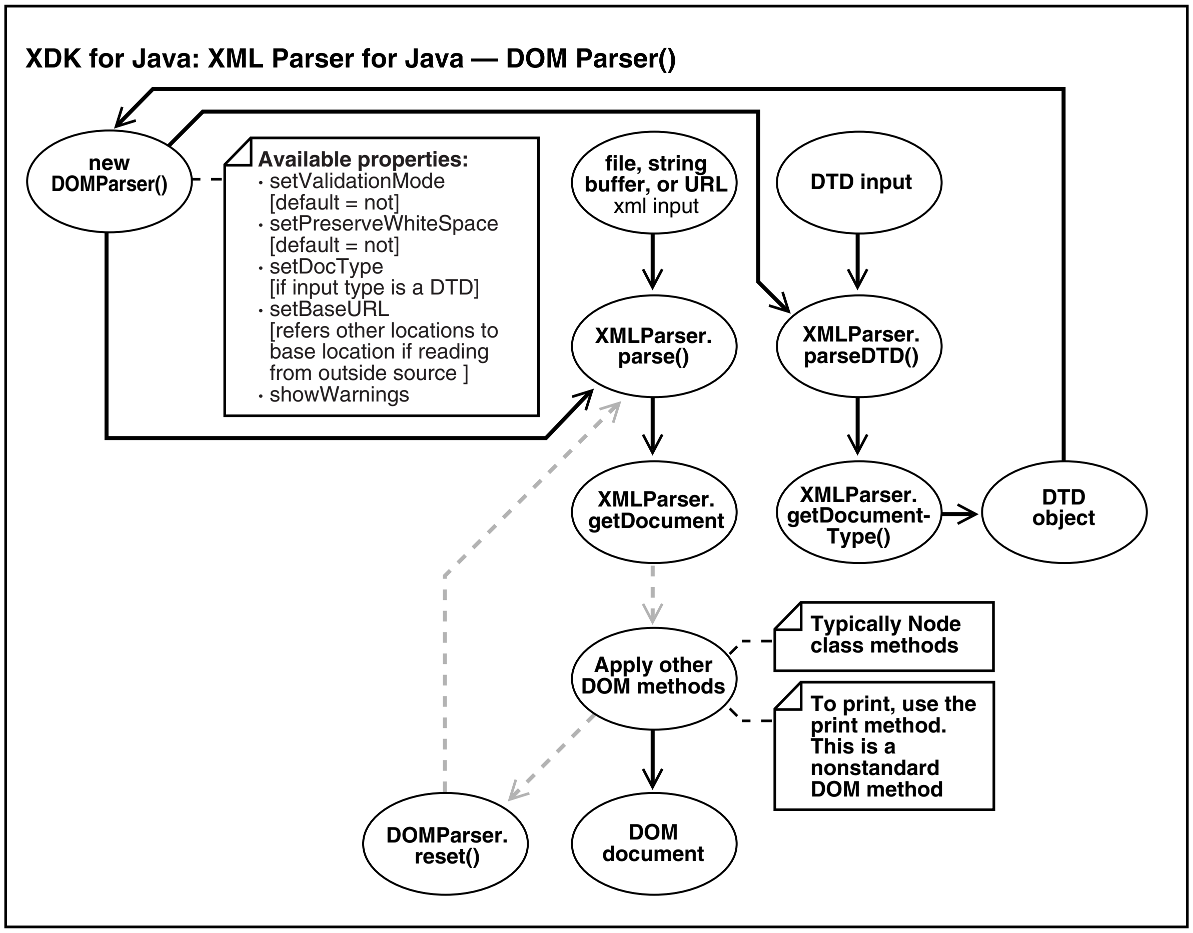
If validation of the input XML document is required, a DTD is used.

1. A new `DOMParser()` is called. The methods to apply to this object are:
   - `setValidateMode()`
   - `setPreserveWhiteSpace()`
   - `setDocType()`

- `setBaseURL()`
- `showWarnings()`

2. The results of `DOMParser()` are passed to `XMLParser.parseDTD()` method along with the DTD input.
3. `XMLParser.getDocumentType()` method sends the resulting DTD object back to the new `DOMParser()` and the process continues until the DTD has been applied.

Figure 3–4 XML Parser for Java: DOMParser()


XDK for Java: XML Parser for Java — DOM Parser()
new DOMParser()
Available properties:
· setValidationMode [default = not]
· setPreserveWhiteSpace [default = not]
· setDocType [if input type is a DTD]
· setBaseURL [refers other locations to base location if reading from outside source]
· showWarnings
file, string buffer, or URL xml input
DTD input
XMLParser. parse()
XMLParser. parseDTD()
XMLParser. getDocument
XMLParser. getDocument-Type()
DTD object
Apply other DOM methods
Typically Node class methods
To print, use the print method. This is a nonstandard DOM method
DOMParser. reset()
DOM document

Comments on Example 1: DOMSample.java

These comments are for Example 1: `DOMSample.java` which follows immediately after this section.

1. Declare a new `DOMParser()` instance:

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

2. The XML input is a URL generated from the input filename:

```
URL url = DemoUtil.createURL(argv[0]);
```

3. The `DOMParser` class has several methods you can use. The example uses:

```
parser.setErrorStream(System.err);
parser.setValidationMode(DTD_VALIDATION);
parser.showWarnings(true);
```

4. The input document is parsed:

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

5. The DOM tree document is obtained:

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

6. This program applies the `node` class methods:

   - `getElementsByTagName()`
   - `getTagName()`
   - `getAttributes()`
   - `getNodeName()`
   - `getNodeValue()`

7. The attributes of each element are printed.

> **Note:** No DTD input is shown in `DOMSample.java`.

XML Parser for Java Example 1: DOMSample.java

This example shows the Java code that uses the preceding steps.

```
/* Copyright (c) Oracle Corporation 2000, 2001. All Rights Reserved. */

/**
 * DESCRIPTION
 * This file demonstates a simple use of the parser and DOM API.
 * The XML file that is given to the application is parsed and the
 * elements and attributes in the document are printed.
 * The use of setting the parser options is demonstrated.
 */

import java.net.URL;

import org.w3c.dom.Node;
import org.w3c.dom.Element;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.NamedNodeMap;

import oracle.xml.parser.v2.DOMParser;
import oracle.xml.parser.v2.XMLDocument;

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);
         }

         // Get an instance of the parser
         DOMParser parser = new DOMParser();

// Generate a URL from the filename.
         URL url = DemoUtil.createURL(argv[0]);

         // Set various parser options: validation on,
         // warnings shown, error stream set to stderr.
         parser.setErrorStream(System.err);
         parser.setValidationMode(DOMParser.DTD_VALIDATION);
         parser.showWarnings(true);

// Parse the document.
         parser.parse(url);

         // Obtain the document.
         XMLDocument doc = parser.getDocument();

         // Print document elements
         System.out.print("The elements are: ");
         printElements(doc);

         // Print document element attributes
         System.out.println("The attributes of each element are: ");
         printElementAttributes(doc);
      }
      catch (Exception e)
      {
         System.out.println(e.toString());
      }
   }

   static void printElements(Document doc)
   {
      NodeList nl = doc.getElementsByTagName("*");
      Node n;

      for (int i=0; i<nl.getLength(); i++)
      {
         n = nl.item(i);
         System.out.print(n.getNodeName() + " ");
      }

      System.out.println();
   }

   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();
            }
         }
      }
```

Using XML Parser for Java: DOMNamespace Class

Figure 3–3 illustrates the main processes involved when parsing an XML document using the DOM interface. The following example illustrates how to use the `DOMNamespace` class:

XML Parser for Java Example 2: Parsing a URL — DOMNamespace.java

See the comments in this source code for a guide to the use of methods. The program begins with these comments:

```
/**
 * DESCRIPTION
 * This file demonstates a simple use of the parser and Namespace
 * extensions to the DOM APIs.
 * The XML file that is given to the application is parsed and the
 * elements and attributes in the document are printed.
 */
```

The methods used on `XMLElement` from the `NSName` interface, which provides Namespace support for element and attribute names, are:

- `getQualifiedName()` returns the qualified name
- `getLocalName()` returns the local name
- `getNamespace()` returns the resolved Namespace for the name
- `getExpandedName()` returns the fully resolved name.

Here is a how they are used later in the code:

```
         // Use the methods getQualifiedName(), getLocalName(), getNamespace()
         // and getExpandedName() in NSName interface to get Namespace
         // information.

         qName = nsElement.getQualifiedName();
         System.out.println("  ELEMENT Qualified Name:" + qName);
         localName = nsElement.getLocalName();
         System.out.println("  ELEMENT Local Name    :" + localName);

         nsName = nsElement.getNamespace();
         System.out.println("  ELEMENT Namespace     :" + nsName);

         expName = nsElement.getExpandedName();
         System.out.println("  ELEMENT Expanded Name :" + expName);
      }
```

For the attributes, the method `getNodeValue()` returns the value of this node, depending on its type. Here is another excerpt from later in this program:

```
nnm = e.getAttributes();

if (nnm != null)
{
   for (i=0; i < nnm.getLength(); i++)
   {
      nsAttr = (XMLAttr) nnm.item(i);

      // Use the methods getExpandedName(), getQualifiedName(),
      // getNodeValue() in NSName
      // interface to get Namespace information.

      attrname = nsAttr.getExpandedName();
      attrqname = nsAttr.getQualifiedName();
      attrval = nsAttr.getNodeValue();
```

No DTD is input is shown in `DOMNameSpace.java`.

Using XML Parser for Java: SAXParser Class

Applications can register a SAX handler to receive notification of various parser events. `XMLReader` is the interface that an XML parser's SAX2 driver must implement. This interface enables an application to set and query features and properties in the parser, to register event handlers for document processing, and to initiate a document parse.

All SAX interfaces are assumed to be synchronous: the parse methods must not return until parsing is complete, and readers must wait for an event-handler callback to return before reporting the next event.

This interface replaces the (now deprecated) SAX 1.0 Parser interface. The `XMLReader` interface contains two important enhancements over the old parser interface:

- It adds a standard way to query and set features and properties.
- It adds Namespace support, which is required for many higher-level XML standards.

Table 3–3 lists the `SAXParser` methods.

Table 3–3 SAXParser Methods

Method	Description
`getContentHandler()`	Returns the current content handler
`getDTDHandler()`	Returns the current DTD handler
`getEntityResolver()`	Returns the current entity resolver
`getErrorHandler()`	Returns the current error handler
`getFeature(java.lang.String name)`	Looks up the value of a feature
`getProperty(java.lang.String name)`	Looks up the value of a property
`setContentHandler(ContentHandler handler)`	Enables an application to register a content event handler
`setDocumentHandler(DocumentHandler handler)`	Deprecated as of SAX2.0; replaced by `setContentHandler()`
`setDTDHandler(DTDHandler handler)`	Enables an application to register a DTD event handler
`setEntityResolver(EntityResolver resolver)`	Enables an application to register an entity resolver
`setErrorHandler(ErrorHandler handler)`	Enables an application to register an error event handler
`setFeature(java.lang.String name, boolean value)`	Sets the state of a feature
`setProperty(java.lang.String name, java.lang.Object value)`	Sets the value of a property

Figure 3–5 shows the main steps for coding with the `SAXParser` class.

1. Create a new handler for the parser:

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

2. Declare a new `SAXParser()` object. Table 3–3 lists all the available methods.

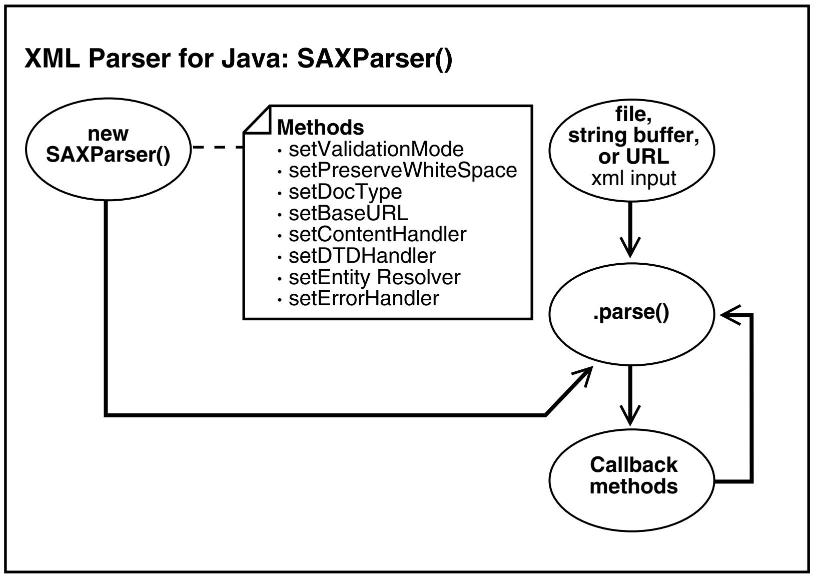
```
Parser parser = new SAXParser;
```

3. Set validation mode as `DTD_VALIDATION`.
4. Convert the input file to URL and parse:

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

5. Parse methods return when parsing completes. Meanwhile the process waits for an event-handler callback to return before reporting the next event.
6. The parsed XML document is available for output by this application. Interfaces used are:
   - `DocumentHandler`
   - `EntityResolver`
   - `DTDHandler`
   - `ErrorHandler`

Figure 3–5 Using SAXParser Class



XML Parser for Java Example 3: Using the Parser and SAX API (SAXSample.java)

This example illustrates how you can use `SAXParser` class and several handler interfaces. See the comments in this source code for a guide to the use of methods.

SAX is a standard interface for event-based XML parsing. The parser reports parsing events directly through callback functions such as `setDocumentLocator()` and `startDocument()`. This application uses handlers to deal with the different events.

```
/* Copyright (c) Oracle Corporation 2000, 2001. All Rights Reserved. */

/**
 * DESCRIPTION
 * This file demonstates a simple use of the parser and SAX API.
 * The XML file that is given to the application is parsed and
 * prints out some information about the contents of this file.
 */

import java.net.URL;

import org.xml.sax.Parser;
import org.xml.sax.Locator;
import org.xml.sax.AttributeList;
import org.xml.sax.HandlerBase;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

import oracle.xml.parser.v2.SAXParser;

public class SAXSample extends HandlerBase
{
   // Store the locator
   Locator locator;

   static public void main(String[] argv)
   {
      try
      {
         if (argv.length != 1)
         {
            // Must pass in the name of the XML file.
            System.err.println("Usage: SAXSample filename");
            System.exit(1);
         }
         // Create a new handler for the parser
         SAXSample sample = new SAXSample();

         // Get an instance of the parser
         Parser parser = new SAXParser();

         // set validation mode
         ((SAXParser)parser).setValidationMode(SAXParser.DTD_VALIDATION);
         // Set Handlers in the parser
         parser.setDocumentHandler(sample);
         parser.setEntityResolver(sample);
         parser.setDTDHandler(sample);
         parser.setErrorHandler(sample);

         // Convert file to URL and parse
         try
         {
            parser.parse(DemoUtil.createURL(argv[0]).toString());
         }
         catch (SAXParseException e)
         {
            System.out.println(e.getMessage());
         }
         catch (SAXException e)
         {
            System.out.println(e.getMessage());
         }
      }
      catch (Exception e)
      {
         System.out.println(e.toString());
      }
   }

   //////////////////////////////////////////////////////////////////////
   // Sample implementation of DocumentHandler interface.
   //////////////////////////////////////////////////////////////////////

   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");
   }
```

```
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);
   }

}

public void endElement(String name) throws SAXException
{
   System.out.println("EndElement:"+name);
}

public void characters(char[] cbuf, int start, int len)
{
   System.out.print("Characters:");
   System.out.println(new String(cbuf,start,len));
}

public void ignorableWhitespace(char[] cbuf, int start, int len)
{
   System.out.println("IgnorableWhiteSpace");
}


public void processingInstruction(String target, String data)
           throws SAXException
{
   System.out.println("ProcessingInstruction:"+target+" "+data);
}



//////////////////////////////////////////////////////////////////////
// Sample implementation of the EntityResolver interface.
//////////////////////////////////////////////////////////////////////


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;
}

//////////////////////////////////////////////////////////////////////
// Sample implementation of the DTDHandler interface.
//////////////////////////////////////////////////////////////////////
```

```
   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);
   }

   //////////////////////////////////////////////////////////////////////
   // Sample implementation of the ErrorHandler interface.
   //////////////////////////////////////////////////////////////////////


   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());
   }
}
```

XML Parser for Java Example 4: (SAXNamespace.java)

See the comments in this source code for use of the SAX APIs.

```
/* Copyright (c) Oracle Corporation 2000, 2001. All Rights Reserved. */

/**
 * DESCRIPTION
 * This file demonstrates a simple use of the Namespace extensions to
 * the SAX 1.0 APIs.
 */

import java.net.URL;

import org.xml.sax.HandlerBase;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

// Extensions to the SAX Interfaces for Namespace support.
import oracle.xml.parser.v2.XMLDocumentHandler;
import oracle.xml.parser.v2.DefaultXMLDocumentHandler;
import oracle.xml.parser.v2.NSName;
import oracle.xml.parser.v2.SAXAttrList;

import oracle.xml.parser.v2.SAXParser;

public class SAXNamespace {

  static public void main(String[] args) {

     String fileName;

     //Get the file name

     if (args.length == 0)
     {
        System.err.println("No file Specified!!!");
        System.err.println("USAGE: java SAXNamespace <filename>");
        return;
     }
     else
     {
        fileName = args[0];
     }


     try {

        // Create handlers for the parser

        // Use the XMLDocumentHandler interface for namespace support
        // instead of org.xml.sax.DocumentHandler
        XMLDocumentHandler xmlDocHandler = new XMLDocumentHandlerImpl();

        // For all the other interface use the default provided by
        // Handler base
        HandlerBase defHandler = new HandlerBase();

        // Get an instance of the parser
        SAXParser parser = new SAXParser();

        // set validation mode
        ((SAXParser)parser).setValidationMode(SAXParser.DTD_VALIDATION);

        // Set Handlers in the parser
        // Set the DocumentHandler to XMLDocumentHandler
        parser.setDocumentHandler(xmlDocHandler);

        // Set the other Handler to the defHandler
        parser.setErrorHandler(defHandler);
        parser.setEntityResolver(defHandler);
        parser.setDTDHandler(defHandler);

        try
        {
           parser.parse(DemoUtil.createURL(fileName).toString());
        }
        catch (SAXParseException e)
        {
           System.err.println(args[0] + ": " + e.getMessage());
        }
        catch (SAXException e)
```

```
         {
            System.err.println(args[0] + ": " + e.getMessage());
         }
      }
      catch (Exception e)
      {
         System.err.println(e.toString());
      }
   }

}

/***********************************************************************
 Implementation of XMLDocumentHandler interface. Only the new
 startElement and endElement interfaces are implemented here. All other
 interfaces are implemented in the class HandlerBase.
**********************************************************************/

class XMLDocumentHandlerImpl extends DefaultXMLDocumentHandler
{

   public void XMLDocumentHandlerImpl()
   {
   }


   public void startElement(NSName name, SAXAttrList atts) throws SAXException
   {

      // Use the methods getQualifiedName(), getLocalName(), getNamespace()
      // and getExpandedName() in NSName interface to get Namespace
      // information.

      String qName;
      String localName;
      String nsName;
      String expName;

      qName = name.getQualifiedName();
      System.out.println("ELEMENT Qualified Name:" + qName);

      localName = name.getLocalName();
      System.out.println("ELEMENT Local Name    :" + localName);

      nsName = name.getNamespace();
      System.out.println("ELEMENT Namespace     :" + nsName);

      expName = name.getExpandedName();
      System.out.println("ELEMENT Expanded Name :" + expName);

      for (int i=0; i<atts.getLength(); i++)
      {

      // Use the methods getQualifiedName(), getLocalName(), getNamespace()
      // and getExpandedName() in SAXAttrList interface to get Namespace
      // information.

         qName = atts.getQualifiedName(i);
         localName = atts.getLocalName(i);
         nsName = atts.getNamespace(i);
         expName = atts.getExpandedName(i);

         System.out.println(" ATTRIBUTE Qualified Name   :" + qName);
         System.out.println(" ATTRIBUTE Local Name       :" + localName);
         System.out.println(" ATTRIBUTE Namespace        :" + nsName);
         System.out.println(" ATTRIBUTE Expanded Name    :" + expName);


         // 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(NSName name) throws SAXException
   {
      // Use the methods getQualifiedName(), getLocalName(), getNamespace()
      // and getExpandedName() in NSName interface to get Namespace
      // information.

      String expName = name.getExpandedName();
      System.out.println("ELEMENT Expanded Name  :" + expName);
   }

}
```

Using the XML Parser for Java

Here are some helpful hints for using the XML Parser for Java. This section contains these topics:

- Using DOM and SAX APIs for Java
- Using Character Sets with the XML Parser for Java
- General Questions About XML Parser for Java

Using DOM and SAX APIs for Java

Here is some further information about the DOM and SAX APIs.

Using the DOM API to Count Tagged Elements

To get the number of elements in a particular tag using the parser, you can use the `getElementsByTagName()` method that returns a node list of all descent elements with a given tag name. You can then find out the number of elements in that node list to determine the number of the elements in the particular tag.

Creating a Node with a Value to Be Set Later

If you check the DOM specification, referring to the table discussing the node type, you will find that if you are creating an element node, its node value is `null`, and cannot be set. However, you can create a text node and append it to the element node. You can then put the value in the text node.

Traversing the XML Tree Using XPATH

You can traverse the tree by using the DOM API. Alternately, you can use the `selectNodes()` method which takes XPath syntax to navigate through the XML document. `selectNodes()` is part of `oracle.xml.parser.v2.XMLNode`.

Finding the First Child Node Element Value

Here is how to efficiently obtain the value of first child node of the element without going through the DOM tree. If you do not need the entire tree, use the SAX interface to return the desired data. Since it is event-driven, it does not have to parse the whole document.

Using the XMLNode.selectNodes() Method

The `selectNodes()` method is used in `XMLElement` and `XMLDocument` nodes. This method is used to extract contents from the tree or subtree based on the select patterns allowed by XSL. The optional second parameter of `selectNodes`, is used to resolve namespace prefixes (that is, it returns the expanded namespace URL given a prefix). `XMLElement` implements `NSResolver`, so it 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 following sample code uses `selectNodes`.

```
public class SelectNodesTest  {
public static void main(String[] args) throws Exception {
String pattern = "/family/member/text()";
String file    = args[0];

if (args.length == 2)
  pattern = args[1];

DOMParser dp = new DOMParser();

dp.parse(createURL(file));  // Include createURL from DOMSample
XMLDocument xd = dp.getDocument();
XMLElement e = (XMLElement) xd.getDocumentElement();
NodeList nl = e.selectNodes(pattern, e);
for (int i = 0; i < nl.getLength(); i++) {
   System.out.println(nl.item(i).getNodeValue());
    }
  }
}

> java SelectNodesTest family.xml
Sarah
Bob
Joanne
Jim

> java SelectNodesTest family.xml //member/@memberid
m1
m2
m3
m4
```

Generating an XML Document from Data in Variables

Here is an example of XML document generation starting from information contained in simple variables, such as when a client fills in a Java form and wants to obtain an XML document.

If you have two variables in Java:

```
String firstname = "Gianfranco";
String lastname = "Pietraforte";
```

The two ways to get this information into an XML document are as follows:

1. Make an XML document in a string and parse it:

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

2. Use DOM APIs to construct the document and append it together:

   ```
   Document xmldoc = new XMLDocument();
   Element e1 = xmldoc.createElement("person");
   xmldoc.appendChild(e1);
   Element e2 = xmldoc.createElement("first");
   e1.appendChild(e2);
   Text t = xmldoc.createText(firstname);
   e2.appendChild(t);
   ```

Using the DOM API to Print Data in the Element Tags

For DOM, `<name>macy</name>` is actually an element named `name` with a child node (`Text Node`) of value `macy`. The sample code is:

```
String value = myElement.getFirstChild().getNodeValue();
```

Building XML Files from Hash Table Value Pairs

If you have a hash table key = value name = george zip = 20000:

```
<key>value</key><name>george</name><zip>20000</zip>
```

1. Get the enumeration of keys from your hash table.
2. Loop while `enum.hasMoreElements()`.
3. For each key in the enumeration, use the `createElement()` on DOM document to create an element by the name of the key with a child text node with the value of the value of the hash table entry for that key.

DOM Exception WRONG_DOCUMENT_ERR on Node.appendChild()

If you have the following code snippet:

```
Document doc1 = new XMLDocument();
Element element1 = doc1.creatElement("foo");
Document doc2 = new XMLDocument();
Element element2 = doc2.createElement("bar");
element1.appendChild(element2);
```

You will get a DOM exception of `WRONG_DOCUMENT_ERR` on calling the `appendChild()` routine, since the owner document of `element1` is `doc1` while that of `element2` is `doc2`. `AppendChild()` only works within a single tree and the example uses two different ones. You need to use `importNode()` or `adoptNode()` instead

Getting DOMException when Setting Node Value

If you create an element node, its `nodeValue` is `null` and hence cannot be set. You get the following error:

```
oracle.xml.parser.XMLDOMException: Node cannot be modified while trying to set
 the value of a newly created node as below:
  String eName="Mynode";
  XMLNode aNode = new XMLNode(eName, Node.ELEMENT_NODE);
  aNode.setNodeValue(eValue);
```

Extracting Embedded XML from a CDATA Section

Here is an example to extract XML from the `CDATA` section of a DTD, which is:

```
<PAYLOAD>
<![CDATA[<?xml version = '1.0' encoding = 'ASCII' standalone = 'no'?>
<ADD_PO_003>
   <CNTROLAREA>
      <BSR>
         <VERB value="ADD">ADD</VERB>
         <NOUN value="PO">PO</NOUN>
         <REVISION value="003">003</REVISION>
      </BSR>
   </CNTROLAREA>
</ADD_PO_003>]]>
</PAYLOAD>
```

Extracting PAYLOAD to do Extra Processing You cannot use a different encoding on the nested XML document included as text inside the `CDATA`, so having the XML declaration of the embedded document seems of little value. If you do not need the XML declaration, then embed the message as real elements into the `<PAYLOAD>` instead of as a text chunk, which is what `CDATA` does for you.

Use the following code:

```
String s = YourDocumentObject.selectSingleNode("/OES_MESSAGE/PAYLOAD");
```

The data is not parsed because it is in a `CDATA` section when you select the value of `PAYLOAD.`

You have asked for it to be a big text chunk, which is what it will give you. You must parse the text chunk yourself (another benefit of not using the `CDATA` approach) this way:

```
YourParser.parse( new StringReader(s));
```

where `s` is the string you got in the previous step.

Using Character Sets with the XML Parser for Java

Here are hints about character sets:

Reading a Unicode XML File

When reading an XML document stored in an operating system file, do not use the `FileReader` class. Instead, use the XML Parser for Java to automatically detect the character encoding of the document. Given a binary input stream with no external encoding information, the parser automatically figures out the character encoding based on the byte order mark and encoding declaration of the XML document. Any well-formed document in any supported encoding can be successfully parsed using the following sample code:

```
import java.io.*;
import oracle.xml.parser.v2.*;
public class I18nSafeXMLFileReadingSample
{
public static void main(String[] args) throws Exception
{
// create an instance of the xml file
File file = new File("myfile.xml");
// create a binary input stream
FileInputStream fis = new FileInputStream(file);
// buffering for efficiency
BufferedInputStream in = new BufferedInputStream(fis);
// get an instance of the parser
DOMParser parser = new DOMParser();
// parse the xml file
parser.parse(in);
}
```

Writing an XML File in UTF-8

`FileWriter` class should not be used in 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 that assumes the default file encoding can cause problems. The following example shows how to avoid these problems:

```
mport java.io.*;
import oracle.xml.parser.v2.*;

public class I18nSafeXMLFileWritingSample
{
  public static void main(String[] args) throws Exception
  {
    // create a test document
    XMLDocument doc = new XMLDocument();
    doc.setVersion("1.0");
    doc.appendChild(doc.createComment("This is a test empty document."));
    doc.appendChild(doc.createElement("root"));

    // create a file
    File file = new File("myfile.xml");

    // create a binary output stream to write to the file just created
    FileOutputStream fos = new FileOutputStream(file);

    // create a Writer that converts Java character stream to UTF-8 stream
    OutputStreamWriter osw = new OutputStreamWriter( fos,"UTF8");

    // buffering for efficiency
    Writer w = new BufferedWriter(osw);

    // create a PrintWriter to adapt to the printing method
    PrintWriter out = new PrintWriter(w);

    // print the document to the file through the connected objects
    doc.print(out);
  }
}
```

Writing Parsing XML Stored in NCLOB with UTF-8 Encoding

The following problem with parsing XML stored in an `NCLOB` column using UTF-8 encoding was reported.

An XML sample that is loaded into the database contains two UTF-8 multibyte characters: The text is supposed to be:

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

A Java stored function was written that uses the default connection object to connect to the database, runs a select query, gets the `OracleResultSet`, calls the `getCLOB()` method and calls the `getAsciiStream()` method on the CLOB object. Then it executes the following code to get the XML into a DOM object:

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

The code throws an exception stating that the XML contains an invalid UTF-8 encoding. The character (`0xc2`, `0x82`) is valid UTF-8. The character can be distorted when `getAsciiStream()` is called.

To solve this problem, use `getUnicodeStream()` and `getBinaryStream()` instead of `getAsciiStream()`.

If this does not work, try to print out the characters to make sure that they are not distorted before they are sent to the parser in step: `parser.parse(istr)`

Parsing a Document Containing Accented Characters

This is the way to parse a document containing accented characters:

```
DOMParser parser=new DOMParser();
parser.setPreserveWhitespace(true);
parser.setErrorStream(System.err);
parser.setValidationMode(false);
parser.showWarnings(true);
parser.parse ( new FileInputStream(new File("PruebaA3Ingles.xml")));
```

Storing Accented Characters in an XML Document

If you have stored accented characters, for example, an é, in your XML file and then attempt to parse the XML file with the XML Parser for Java, the parser may throw the following exception:

```
'Invalid UTF-8 encoding'
```

You can read in accented characters in their hex or decimal format within the XML document, for example:

```
&#xe9;
```

but if you prefer not to do this, set the encoding based on the character set you were using when you created the XML file. Try setting the encoding to ISO-8859-1 (Western

European ASCII). Use that encoding or something different, depending on the tool or operating system you are using.

If you explicitly set the encoding to UTF-8 (or do not specify it at all), the parser interprets your accented character (which has an ASCII value > 127) as the first byte of a UTF-8 multibyte sequence. If the subsequent bytes do not form a valid UTF-8 sequence, you get an error.

This error just means that your editor is not saving the file with UTF-8 encoding. For example, it might be saving it with ISO-8859-1 encoding. The encoding is a particular scheme used to write the Unicode character number representation to disk. Just adding this string to the top of the document does not cause your editor to write out the bytes representing the file to disk using UTF-8 encoding:

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

Notepad uses UTF-8 on Windows systems.

You Cannot Dynamically Set the Encoding for an Input XML File

You need to include the proper encoding declaration in your document according to the specification. You cannot use `setEncoding()` to set the encoding for your input document. `SetEncoding()` is used with `oracle.xml.parser.v2.XMLDocument` to set the correct encoding for the printing.

Using System.out.println() and Special Characters

You cannot use `System.out.println()`. You need to use an output stream which is encoding aware (for example, `OutputStreamWriter`). You can construct an OutputStreamWriter and use the `write(char[], int, int)` method to print.

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

General Questions About XML Parser for Java

These are general questions:

Including Binary Data in an XML Document

There is no way to directly include binary data within the document; however, there are two ways to work around this:

- Binary data can be referenced as an external unparsed entity that resides in a different file.
- Binary data can be uuencoded (meaning converted into ASCII data by `UUENCODE` program) and be included in a `CDATA` section. The limitation on the encoding technique is to ensure that it only produces legal characters for the `CDATA` section.
- base64 is a command line utility which encodes and decodes files in a format used by MIME-encoded documents.

Displaying an XML Document

If you are using IE5 as your browser you can display the XML document directly. Otherwise, you can use the Oracle XSLT Processor version 2 to create the HTML document using an XSL Stylesheet. The XDK JavaBeans also enable you to view your XML document.

Including an External XML File in Another XML File

IE 5.0 will parse an XML file and show the parsed output. Just load the file as you load an HTML page.

The following works, both browsing it in IE5 as well as parsing it with the XML Parser for Java:

```
File: a.xml
<?xml version="1.0" ?>
<!DOCTYPE a [<!ENTITY b SYSTEM "b.xml">]>
 <a>&b;</a>

File: b.xml
 <ok/>
```

When you browse and parse `a.xml` you get the following:

```
<a>
  <ok/>
</a>
```

You Do Not Need Oracle9*i* or Higher to Run XML Parser for Java

XML Parser for Java can be used with any of the supported version Java VMs. The only difference with Oracle9*i* or higher, is that you can load it into the database and use Oracle9*i* JVM which is an internal JVM. For other database versions or servers, you simply run it in an external JVM and as necessary connect to a database through JDBC.

Inserting Characters <, >, ', ", and & into XML Documents

You must use the *entity references:*

- `&gt;` for greater than (>)
- `&lt;` for less than (<)
- `&apos;` for an apostrophe or a single quote (')
- `&quot;` for straight double quotes (")
- `&amp;` for ampersand (&)

Invalid Special Characters in Tags

If you have a tag in XML `<COMPANYNAME>` and use `A&B`, the parser gives an error with invalid character.

Special characters such as `&`, `$`, and `#`, and so on are not allowed to be used. If you are creating an XML document from scratch, you can use a workaround by using only valid `NameChars`. For example, `<A_B>, <AB>, <A_AND_B>` and so on. They are still readable.

If you are generating XML from external data sources such as database tables, then this is a problem which XML 1.0 does not address.

The datatype `XMLType` addresses this problem by offering a function which maps SQL names to XML names. The SQL to XML name mapping function will escape invalid XML `NameChar` in the format of `_XHHHH_` where `HHHH` is a Unicode value of the invalid character. For example, table name `V$SESSION` will be mapped to XML name `V_X0024_SESSION`.

Finally, escaping invalid characters is a workaround to give people a way to serialize names so that they can reload them somewhere else.

Parsing XML from Data of Type String

Currently there is no method that can directly parse an XML document contained within a string. You need to convert the string into an `InputStream` or `InputSource` before parsing. An easy way is to create a `ByteArrayInputStream` using the bytes in the string. For example:

```
/* xmlDoc is a String of xml */
byte aByteArr [] = xmlDoc.getBytes();
ByteArrayInputStream bais = new ByteArrayInputStream (aByteArr, 0,
aByteArr.length);
domParser.parse(bais);
```

Extracting Data from an XML Document into a String

Here is an example to do this:

```
XMLDocument Your Document;
/* Parse and Make Mods */
:
StringWriter sw = new StringWriter();
PrintWriter  pw = new PrintWriter(sw);
YourDocument.print(pw);
String YourDocInString = sw.toString();
```

Illegal Characters in XML Documents

If you limit it to 8-bit, then `#x0-#x8`; `#xB`, `#xC`, `#xE`, and `#xF` are not legal.

Using Entity References with the XML Parser for Java

If the XML Parser for Java does not expand entity references, such as `&[whatever]` and instead, all values are `null`, how can you fix this?

You probably have a simple error defining or using your entities, since Oracle has regression tests that handle entity references without error. A simple example is: `]> Alpha`, then `&status`.

Merging XML Documents

This is done either using DOM or XSLT.

See Also: "Merging Two XML Documents" on page 4-11

The XML Parser for Java Does Not Need a Utility to View the Parsed Output

The parsed external entity only needs to be a well-formed fragment. The following program (with `xmlparser.jar` from version 1) in your `CLASSPATH` shows parsing and printing the parsed document. It's parsing here from a string but the mechanism is no different for parsing from a file, given its URL.

```
import oracle.xml.parser.*;
import java.io.*;
import java.net.*;
import org.w3c.dom.*;
import org.xml.sax.*;
/*
** Simple Example of Parsing an XML File from a String
** and, if successful, printing the results.
**
** Usage: java ParseXMLFromString <hello><world/></hello>
*/
```

```
public class ParseXMLFromString {
  public static void main( String[] arg ) throws IOException, SAXException {
    String theStringToParse =
       "<?xml version='1.0'?>"+
       "<hello>"+
       "  <world/>"+
       "</hello>";
    XMLDocument theXMLDoc = parseString( theStringToParse );
    // Print the document out to standard out
    theXMLDoc.print(System.out);
  }
  public static XMLDocument parseString( String xmlString ) throws
   IOException, SAXException {
   XMLDocument theXMLDoc     = null;
    // Create an oracle.xml.parser.v2.DOMParser to parse the document.
    XMLParser theParser = new XMLParser();
    // Open an input stream on the string
    ByteArrayInputStream theStream =
         new ByteArrayInputStream( xmlString.getBytes() );
    // Set the parser to work in non-Validating mode
    theParser.setValidationMode(DTD_validation);
    try {
      // Parse the document from the InputStream
      theParser.parse( theStream );
      // Get the parsed XML Document from the parser
      theXMLDoc = theParser.getDocument();
    }
    catch (SAXParseException s) {
      System.out.println(xmlError(s));
      throw s;
    }
    return theXMLDoc;
  }
  private static String xmlError(SAXParseException s) {
     int lineNum = s.getLineNumber();
     int  colNum = s.getColumnNumber();
     String file = s.getSystemId();
     String  err = s.getMessage();
     return "XML parse error in file " + file +
            "\n" + "at line " + lineNum + ", character " + colNum +
            "\n" + err;
  }
}
```

Support for Hierarchical Mapping

About the relational mapping of parsed XML data: some users prefer hierarchical storage of parsed XML data. Will `XMLType` address this concern?

Many customers initially have this concern. It depends on what kind of XML data you are storing. If you are storing XML datagrams that are really just encoding of relational information (for example, a purchase order), then you will get much better performance and much better query flexibility (in SQL) by storing the data contained in the XML documents in relational tables, then reproduce on-demand an XML format when any particular data needs to be extracted.

If you are storing documents that are mixed-content, like legal proceedings, chapters of a book, reference manuals, and so on, then storing the documents in chunks and searching them using Oracle Text's XML search capabilities is the best bet.

The book, *Building Oracle XML Applications,* by Steve Muench, covers both of these storage and searching techniques with lots of examples.

Support for Ambiguous Content Mode

Are there plans to add an ambiguous content mode to the XDK Parser for Java?

The XML Parser for Java implements all the XML 1.0 standard, and the XML 1.0 standard requires XML documents to have unambiguous content models. Therefore, there is no way a compliant XML 1.0 parser can implement ambiguous content models.

See Also: http://www.xml.com/axml/target.html#determinism

Generating an XML Document Based on Two Tables

If you want to generate an XML document based on two tables with a master detail relationship. Suppose you have two tables:

- PARENT with columns: ID and PARENT_NAME (Key = ID)
- CHILD with columns: PARENT_ID, CHILD_ID, CHILD_NAME (Key = PARENT_ID + CHILD_ID)

There is a master detail relationship between `PARENT` and `CHILD`. How can you generate a document that looks like this?

```
<?xml version = '1.0'?>
  <ROWSET>
     <ROW num="1">
       <parent_name>Bill</parent_name>
         <child_name>Child 1 of 2</child_name>
         <child_name>Child 2 of 2</child_name>
      </ROW>
      <ROW num="2">
       <parent_name>Larry</parent_name>
         <child_name>Only one child</child_name>
      </ROW>
  </ROWSET>
```

Use an object view to generate an XML document from a master-detail structure. In your case, use the following code:

```
create type child_type is object
(child_name <data type child_name>) ;
/
create type child_type_nst
is table of child_type ;
/

create view parent_child
as
select p.parent_name
, cast
  ( multiset
    ( select c.child_name
      from   child c
      where  c.parent_id = p.id
    ) as child_type_nst
  ) child_type
from parent p
/
```

A `SELECT * FROM parent_child`, processed by an SQL to XML utility generates a valid XML document for your parent child relationship. The structure does not look like the one you have presented, though. It looks like this:

```
<?xml version = '1.0'?>
<ROWSET>
   <ROW num="1">
      <PARENT_NAME>Bill</PARENT_NAME>
      <CHILD_TYPE>
         <CHILD_TYPE_ITEM>
            <CHILD_NAME>Child 1 of 2</CHILD_NAME>
         </CHILD_TYPE_ITEM>
         <CHILD_TYPE_ITEM>
            <CHILD_NAME>Child 2 of 2</CHILD_NAME>
         </CHILD_TYPE_ITEM>
      </CHILD_TYPE>
  </ROW>
   <ROW num="2">
      <PARENT_NAME>Larry</PARENT_NAME>
      <CHILD_TYPE>
         <CHILD_TYPE_ITEM>
            <CHILD_NAME>Only one child</CHILD_NAME>
         </CHILD_TYPE_ITEM>
      </CHILD_TYPE>
  </ROW>
</ROWSET>
```

Using JAXP

The Java API for XML Processing (JAXP) enables you to use the SAX, DOM, and XSLT processors from your Java application. JAXP enables applications to parse and transform XML documents using an API that is independent of a particular XML processor implementation.

JAXP has a *pluggability* layer that enables you to plug in an implementation of a processor. The JAXP APIs have an API structure consisting of abstract classes providing a thin layer for parser pluggability. Oracle has implemented JAXP based on the Sun Microsystems reference implementation.

The sample programs `JAXPExamples.java` and `ora.ContentHandler.java` in the directory `xdk/demo/java/parser/jaxp` demonstrate various ways that the JAXP API can be used to transform any one of the classes of the interface `Source`:

- `DOMSource` class
- `StreamSource` class
- `SAXSource` class

into any one of the classes of the interface `Result`:

`DOMResult` class

`StreamResult` class

`SAXResult` class

These transformations use XML documents as sample input, optional stylesheets as input, and, optionally, a `ContentHandler` class defined in the file `oraContentHandler.java`. For example, one method, `identity`, does an identity transformation where the output XML document is the same as the input XML document. Another method, `xmlFilterChain()`, applies three stylesheets in a chain.

Among the drawbacks of JAXP are the additional interface cost, features that are behind "native" Parsers, and the fact that a DOM cannot be shared by processing components.

> **See Also:** More examples can be found at:
>
> - `http://www.oracle.com/technology/tech/xml`
> - `http://java.sun.com/xml/jaxp/`
> - and in the directory `xdk/demo/java/parser/jaxp`

oraxml: XML Parser for Java Command-line

`oraxml` is a command-line interface to parse an XML document. It checks for well-formedness and validity.

To use `oraxml` ensure that the following is true:

- Your `CLASSPATH` environment variable is set to point to the `xmlparserv2.jar` file that comes with XML V2 Parser for Java. Because `oraxml` supports schema validation, include `xschema.jar` also in your `CLASSPATH`.
- Your `PATH` environment variable can find the Java interpreter that comes with the JDK that you are using.

Table 3–4 lists the `oraxml` command line options.

Table 3–4 oraxml: Command Line Options

Option	Purpose
`-comp` *`fileName`*	Compresses the input XML file
`-decomp` *`fileName`*	Decompresses the input compressed file
`-dtd` *`fileName`*	Validates the input file with DTD Validation
`-enc` *`fileName`*	Prints the encoding of the input file
`-help`	Prints the help message
`-log` *`logfile`*	Writes the errors to the output log file
`-novalidate` *`fileName`*	Checks whether the input file is well-formed
`-schema` *`fileName`*	Validates the input file with Schema Validation
`-version`	Prints the release version
`-warning`	Show warnings

4 XSLT Processor for Java

This chapter contains these topics:

XSLT Processor for Java Overview

Oracle provides **eXtensible Stylesheet Language Transformation (XSLT)** processing for Java, C, C++, and PL/SQL. This chapter focuses on the XSLT Processor for Java. XSLT is a W3C Internet standard that has a version 1.0, and also a 2.0 version currently in process. XSLT also uses XPath, which is the navigational language used by XSLT and has corresponding versions. The XSLT Processor for Java implements both the XSLT and XPath 1.0 standards as well as a draft of the XSLT and XPath 2.0 standard. Please see the `README` for the specific versions.

While XSLT is a function-based language that generally requires a DOM of the input document and stylesheet to perform the transformation, the Java implementation uses SAX, a stream-based parser to create a stylesheet object to perform transformations with higher efficiency and less resources. This stylesheet object can be reused to transform multiple documents without re-parsing the stylesheet.

The XSLT Processor for Java includes additional high performance features. It is thread-safe to allow processing multiple files with a single XSLT Processor for Java and stylesheet object. It is also safe to use clones of the document instance in multiple threads.

Using XSLT Processor for Java

The XSLT Processor for Java operates on two inputs: the XML document to transform, and the XSLT stylesheet that is used to apply transformations on the XML. Each of these two can actually be multiple inputs. One stylesheet can be used to transform multiple XML inputs. Multiple stylesheets can be mapped to a single XML input.

To implement the XSLT Processor in the XML Parser for Java use the `XSLProcessor` class.

Figure 4–1 shows the overall process used by the `XSLProcessor` class. Here are the steps:

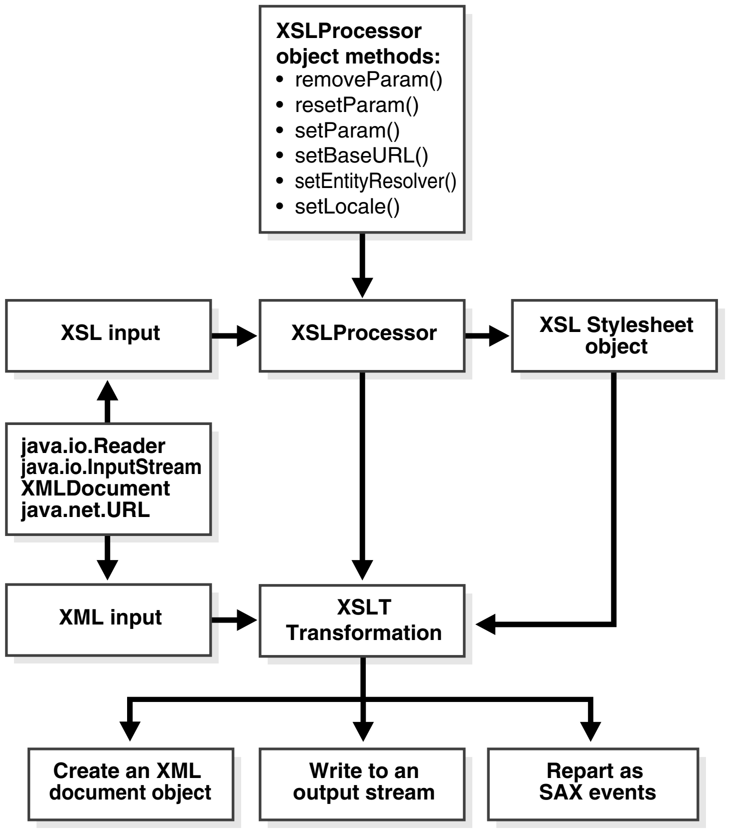
1. Create an `XSLProcessor` object and then use methods from the following list in your Java code. Some of the available methods are:
   - `removeParam()` - remove parameter
   - `RESETPARAM()` – remove all parameters
   - `setParam()` - set parameters for the transformation
   - `setBaseURL()` - set a base URL for any relative references in the stylesheet
   - `setEntityResolver()` - set an entity resolver for any relative references in the stylesheet
   - `setLocale()` - set locale for error reporting
2. Use one of the following input parameters to the method `XSLProcessor.newXSLStylesheet()` to create a stylesheet object:
   - `java.io.Reader`
   - `java.io.InputStream`
   - `XMLDocument`
   - `java.net.URL`

   This creates a stylesheet object that is thread-safe and can be used in multiple XSL Processors.
3. Create a DOM object by passing one of the XML inputs in step 2, to the DOM parser and creating an XML input object with `parser.getDocument`.
4. Your XML inputs and the stylesheet object are input (each using one of the input parameters listed in 2) to the XSL Processor:

   ```
   XSLProcessor.processXSL(xslstylesheet, xml instance)
   ```

   The results of the XSL Transformation can be one of the following:
   - Create an XML document object
   - Write to an output stream
   - Report as SAX events

Figure 4–1 Using XSL Processor for Java



Unlike in HTML, in XML every start tag must have an ending tag and that the tags are case sensitive.

XSLT Processor for Java Example

This example has many comments. It uses one XML document and one XSL stylesheet as inputs.

```
public class XSLSample
{
   public static void main(String args[]) throws Exception
   {
      if (args.length < 2)
      {
         System.err.println("Usage: java XSLSample xslFile xmlFile.");
         System.exit(1);
      }

      // Create a new XSLProcessor.
      XSLProcessor processor = new XSLProcessor();

      // Register a base URL to resolve relative references
      // processor.setBaseURL(baseURL);

      // Or register an org.xml.sax.EntityResolver to resolve
      // relative references
      // processor.setEntityResolver(myEntityResolver);
      // Register an error log
      // processor.setErrorStream(new FileOutputStream("error.log"));

      // Set any global paramters to the processor
      // processor.setParam(namespace, param1, value1);
      // processor.setParam(namespace, param2, value2);

      // resetParam is for multiple XML documents with different parameters

      String xslFile = args[0];
      String xmlFile = args[1];

      // Create a XSLStylesheet
      //  The stylesheet can be created using one of following inputs:
      //
      // XMLDocument xslInput = /* using DOMParser; see later in this code */
      // URL         xslInput = new URL(xslFile);
      // Reader      xslInput = new FileReader(xslFile);

      InputStream xslInput = new FileInputStream(xslFile);

      XSLStylesheet stylesheet = processor.newXSLStylesheet(xslInput);

      // Prepare the XML instance document
      //   The XML instance can be given to the processor in one of
      // following ways:
      //
      // URL         xmlInput = new URL(xmlFile);
      // Reader      xmlInput = new FileReader(xmlFile);
      // InputStream xmlInput = new FileInputStream(xmlFile);
      // Or using DOMParser

      DOMParser parser = new DOMParser();
      parser.retainCDATASection(false);
      parser.setPreserveWhitespace(true);
      parser.parse(xmlFile);
      XMLDocument xmlInput = parser.getDocument();

      // Transform the XML instance
      //   The result of the transformation can be one of the following:
      //
      // 1. Return a XMLDocumentFragment
      // 2. Print the results to a OutputStream
      // 3. Report SAX Events to a ContentHandler

      // 1. Return a XMLDocumentFragment
      XMLDocumentFragment result;
      result = processor.processXSL(stylesheet, xmlInput);

      // Print the result to System.out
      result.print(System.out);

      // 2. Print the results to a OutputStream
      // processor.processXSL(stylesheet, xmlInput, System.out);

      // 3. Report SAX Events to a ContentHandler
      // ContentHandler cntHandler = new MyContentHandler();
      // processor.processXSL(stylesheet, xmlInput, cntHandler);
```

```
    }
}
```

See Also:

- http://www.w3.org/TR/xslt which is the W3C Web site
- http://www.w3.org/style/XSL/ for more information
- "SAX: Event-Based API" on page 3-8

XSLT Command-Line Interface: oraxsl

`oraxsl` is a command-line interface used to apply a stylesheet on multiple XML documents. It accepts a number of command-line options that determine its behavior. `oraxsl` is included in the `$ORACLE_HOME/bin` directory. To use `oraxsl` ensure the following:

- Your `CLASSPATH` environment variable is set to point to the `xmlparserv2.jar` file that comes with XML Parser for Java, version 2.
- Your `PATH` environment variable can find the Java interpreter that comes with JDK 1.2 or higher.

Use the following syntax to invoke `oraxsl`:

```
oraxsl options source stylesheet result
```

`oraxsl` expects to be given a stylesheet, an XML file to transform, and optionally, a result file. If no result file is specified, it outputs the transformed document to the standard output. If multiple XML documents need to be transformed by a stylesheet, use the `-l` or `-d` options in conjunction with the `-s` and `-r` options. These and other options are described in Table 4–1.

Table 4–1 oraxsl: Command Line Options

Option	Purpose
`-d` *`directory`*	Directory with files to transform (the default behavior is to process all files in the directory). If only a certain subset of the files in that directory, for example, one file, need to be processed, this behavior must be changed by using `-l` and specifying just 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).
`-debug`	Debug mode (by default, debug mode is turned off).
`-e` *`error_log`*	The file to write errors and warnings into.
`-h`	Help mode (prints `oraxsl` invocation syntax).
`-i` *`source_extension`*	Extensions to include (used in conjunction with `-d`. Only files with the specified extension are selected).
`-l` *`xml_file_list`*	List of files to transform (enables you to explicitly list the files to be processed).
`-o` *`result_directory`*	Directory to place results (this must be used in conjunction with the `-r` option).
`-p` *`param_list`*	List of Parameters.

Table 4–1 (Cont.) oraxsl: Command Line Options

Option	Purpose
`-r` *`result_extension`*	Extension to use for results (if `-d` or `-l` is specified, this option must be specified to specify the extension to be used for the results of the transformation. So, if you specify the extension "`out`", an input document "`input_doc`" is transformed to "`input_doc.out`". By default, the results are placed in the current directory. This can be changed by using the `-o` option which enables you to specify a directory to hold the results).
`-s` *`stylesheet`*	Stylesheet to use (if `-d` or `-l` is specified, this option needs to be specified to specify the stylesheet to be used. The complete path must be specified).
`-t` *`num_of_threads`*	Number of threads to use for processing (using multiple threads can provide performance improvements when processing multiple documents).
`-v`	Verbose mode (some debugging information is printed and can help in tracing any problems that are encountered during processing).
`-w`	Show warnings (by default, warnings are turned off).
`-x` *`source_extension`*	Extensions to exclude, used in conjunction with `-d`. All files with the specified extension not selected.

XML Extension Functions for XSLT Processing

XML extension functions for XSLT processing allow users of XSLT processor for Java to call any Java method from XSL expressions.

While these are Oracle extensions, the XSLT 1.0 standard provides for implementation-defined extension functions. Stylesheets using these functions may not be interoperable when run on different processors.The functions are language and implementation specific.

This section contains these topics:

- XSLT Processor for Java Extension Functions and Namespaces
- Static Versus Non-Static Methods
- Constructor Extension Function
- Return Value Extension Function
- Datatypes Extension Function
- XSLT Processor for Java Built-In Extensions: ora:node-set and ora:output

XSLT Processor for Java Extension Functions and Namespaces

Java extension functions belong to the namespace that starts with the following:

```
http://www.oracle.com/XSL/Transform/java/
```

An extension function that belongs to the following namespace refers to methods in class `classname`:

```
http://www.oracle.com/XSL/Transform/java/classname
```

For example, the following namespace can be used to call `java.lang.String` methods from XSL expressions:

```
http://www.oracle.com/XSL/Transform/java/java.lang.String
```

Static Versus Non-Static Methods

If the 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 on to the method.

If the extension function is a static method, then all the parameters of the extension function are passed on as parameters to the static function.

XML Parser for Java - XSL Example 1: Static function

The following XSL, static function example prints out '13':

```
<xsl:stylesheet
  xmlns:math="http://www.oracle.com/XSL/Transform/java/java.lang.Math">
  <xsl:template match="/">
  <xsl:value-of select="math:ceil('12.34')"/>
</xsl:template>
</xsl:stylesheet>
```

Note: The XSL class loader only knows about statically added JARs and paths in the `CLASSPATH` - and those specified by `wrapper.classpath`.

Constructor Extension Function

The extension function `new` creates a new instance of the class and acts as the constructor.

XML Parser for Java - XSL Example 2: Constructor Extension Function

The following constructor function example prints out 'HELLO WORLD':

```
<xsl:stylesheet
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>
```

Return Value Extension Function

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`

They can be stored in variables or passed onto 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.

XML Parser for Java XSL- XSL Example 3: Return Value Extension Function

Here is an XSL example illustrating the Return Value Extension function:

```
<!-- Declare extension function namespace -->
<xsl:stylesheet xmlns:parser =
 "http://www.oracle.com/XSL/Transform/java/oracle.xml.parser.v2.DOMParser"
 xmlns:document =
 "http://www.oracle.com/XSL/Transform/java/oracle.xml.parser.v2.XMLDocument" >

<xsl:template match ="/"> <!-- Create a new instance of the parser, store it in
 myparser variable -->
<xsl:variable name="myparser" select="parser:new()"/>
<!-- Call a non-static method of DOMParser. Since the method is a non-static
 method, the first parameter is the instance on which the method is called. This
 is equivalent to $myparser.parse('test.xml') -->
<xsl:value-of select="parser:parse($myparser, 'test.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,'elementname')">
...
</xsl:for-each> </xsl:template>
</xsl:stylesheet>
```

Datatypes Extension Function

Overloading based on number of parameters and type is supported. Implicit type conversion is done between the five XSL types as defined in XSL. Type conversion is done implicitly between (`String, Number, Boolean, ResultTree`) and from `NodeSet` to (`String`, `Number`, `Boolean`, `ResultTree`). Overloading based on two types which can be implicitly converted to each other is not permitted.

XML Parser for Java - XSL Example 4: Datatype Extension Function

The following overloading results in an error in XSL, since `String` and `Number` can be implicitly converted to each other:

- `abc(int i){}`
- `abc(String s){}`

Mapping between XSL type and Java type is done as follows:

```
String ->     java.lang.String
Number ->     int, float, double
Boolean ->    boolean
NodeSet ->    XMLNodeList
ResultTree -> XMLDocumentFragment
```

XSLT Processor for Java Built-In Extensions: ora:node-set and ora:output

Here are the definitions of these Oracle XSL extensions; both are preceded by `xmlns:ora="http://www.oracle.com/XSL/Transform/java"`.

ora:output

This element can be used as a top-level element similar to `xsl:output`. It can have all of the attributes of `xsl:output`, with similar functionality. It has an additional attribute `name`, used as an identifier. When `ora:output` is used in a template, it can only have the attributes `use` and `href`. `use` specifies the top-level `ora:output` to be used, and `href` gives the output URL

ora:node-set

This built-in extension function converts a result tree fragment into a node-set.

Example of Use of Oracle XSL Extensions

The following example illustrates use of both `ora:node-set` and `ora:output`.

If you enter:

```
$ oraxsl foo.xml slides.xsl toc.html
```

where `foo.xml` is any input XML file. You get as output:

- A `toc.html` slide file with a table of contents
- A `slide01.html` file with slide 1
- A `slide02.html` file with slide 2

```
<!--
    | Illustrate using ora:node-set and ora:output
    |
    | Both extensions depend on defining a namespace
    | with the uri of "http://www.oracle.com/XSL/Transform/java"
+-->
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:ora="http://www.oracle.com/XSL/Transform/java">

<!-- <xsl:output> affects the primary result document -->
<xsl:output mode="html" indent="no"/>

<!--
     |   <ora:output> at the top-level enables all attributes
     |   that <xsl:output> enables, but you must provide the
     |   additional "name" attribute to assign a name to
     |   these output settings to be used later.
+-->
<ora:output name="myOutput" mode="html" indent="no"/>
<!--
     | This top-level variable is a result-tree fragment
+-->
<xsl:variable name="fragment">
   <slides>
       <slide>
          <title>First Slide</title>
             <bullet>Point One</bullet>
             <bullet>Point Two</bullet>
             <bullet>Point Three</bullet>
       </slide>
       <slide>
          <title>Second Slide</title>
              <bullet>Point One</bullet>
              <bullet>Point Two</bullet>
              <bullet>Point Three</bullet>
       </slide>
   </slides>
</xsl:variable>
<xsl:template match="/">
<!--    | We cannot "de-reference" a result-tree-fragment to
        | navigate into it with an XPath expression. However, using
        | the ora:node-set() built-in extension function, you can
        | "cast" a result-tree fragment to a node-set which *can*
        | then be navigated using XPath. Since we'll use the node-set
        | of <slides> twice later, we save the node-set in a variable.
+-->
<xsl:variable name="slides" select="ora:node-set($fragment)"/>
<!--
    | This <html> page will go to the primary result document.
    | It is a "table of contents" for the slide show, with
    | links to each slide. The "slides" will each be generated
    | into *secondary* result documents, each slide having
    | a file name of "slideNN.html" where NN is the two-digit
    | slide number
+-->
<html>
   <body>
      <h1>List of All Slides</h1>
<xsl:apply-templates select="$slides" mode="toc"/>
   </body>
</html>
<!--
     | Now go apply-templates to format each slide
+-->
<xsl:apply-templates select="$slides"/>
</xsl:template>
<!-- In 'toc' mode, generate a link to each slide we match -->
<xsl:template match="slide" mode="toc">
     <a href="slide{format-number(position(),'00')}.html">
<xsl:value-of select="title"/>
     </a><br/>
</xsl:template>
<!--
      | For each slide matched, send the output for the current
      | <slide> to a file named "slideNN.html". Use the named
      | output style defined earlier called "myOutput".
<xsl:template match="slide">
<ora:output use="myOutput href="slide{format-number(position(),'00')}.html">
<html>
    <body>
<xsl:apply-templates select="title"/>
       <ul>
<xsl:apply-templates select="*[not(self::title)]"/>
       </ul>
    </body>
</html>
</ora:output>
</xsl:template>
<xsl:template match="bullet">
    <li><xsl:value-of select="."/></li>
</xsl:template>
<xsl:template match="title">
    <h1><xsl:value-of select="."/></h1>
</xsl:template>
```

```
</xsl:stylesheet>
```

Hints for Using the XSLT Processor for Java and XSL

This section lists XSL and XSLT Processor for Java hints, and contains these topics:

- Merging Two XML Documents
- Extracting Embedded XML Using Only XSLT
- Support of Output Method "html" in the XSL Parser
- Creating an HTML Input Form
- Correct XSL Namespace URI
- XSL Processor Produces Multiple Outputs
- Keeping White Spaces in Your Output
- XDK Utilities That Translate Data from Other Formats to XML
- Multiple Threads Using a Single XSLProcessor and Stylesheet
- Using Document Clones in Multiple Threads
- Disabling Output Escaping Is Supported

Merging Two XML Documents

To merge two XML documents, you can either use the DOM APIs or use XSLT-based approaches.

If you use the DOM APIs, then you have to copy the DOM node from the source DOM document before you can append it to the destination DOM document. This operation is required to avoid DOM document ownership errors, like `WRONG_DOCUMENT_ERR`. Both the `importNode()` method, introduced in DOM 2, and `adoptNode()` method, introduced in DOM 3, can be used to copy and paste a DOM document fragment or a DOM node across different XML documents.

Example: Using importNode() from DOM Level 2

```
Document doc1 = new XMLDocument();
Element element1 = doc1.createElement("foo");
Document doc2 = new XMLDocument();
Element element2 = doc2.createElement("bar");
element2 = doc1.importNode(element2);
element1.appendChild(element2);
```

Example: Using adoptNode from DOM Level 3

```
Document doc1 = new XMLDocument();
Element element1 = doc1.createElement("foo");
Document doc2 = new XMLDocument();
Element element2 = doc2.createElement("bar");
element2 = doc1.adoptNode(element2);
element1.appendChild(element2);
```

The difference between using `adoptNode()` and `importNode()` is that using `adoptNode()`, the source DOM node is removed from the original DOM document, while using `importNode()`, the source node is not altered or removed.

If the merging operation is simple, you can also use the XSLT-based approaches. For example, you have two XML documents such as:

Example: demo1.xml

```
<messages>
   <msg>
      <key>AAA</key>
      <num>01001</num>
   </msg>
   <msg>
      <key>BBB</key>
      <num>01011</num>
    </msg>
</messages>
```

Example: demo2.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>
```

Here is an example stylesheet, that merges the two XML documents, `demo1.xml` and `demo2.xml`, based on matching the `<key/>` element values.

Example: demomerge.xsl

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output indent="yes"/>
<xsl:variable name="doc2" select="document('demo2.xml')"/>
<xsl:template match="@*|node()">
   <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
   </xsl:copy>
</xsl:template>

<xsl:template match="msg">
   <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
         <text><xsl:value-of select="$doc2/messages/msg[key=current()/key]/text"/>
         </text>
   </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Enter the following at the command line:

```
$ oraxsl demo1.xml demomerge.xsl
```

Then, you get the following merged result:

```
<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 method is obviously not as efficient for larger files as an equivalent database join of two tables, but this illustrates the technique if you have only XML files to work with.

Extracting Embedded XML Using Only XSLT

The content of your `CDATA`, it is just text. If you want the text content to be output without escaping the angle-brackets:

```
<xsl:value-of select="/OES_MESSAGE/PAYLOAD" disable-output-escaping="yes"/>
```

Support of Output Method "html" in the XSL Parser

XSLT fully supports all options of `<xsl:output>`. Your XSL stylesheet must be a well-formed XML document. Instead of using the `<BR>` element, you must use `<BR/>`. The `<xsl:output method="html"/>` requests that when the XSLT engine writes out the result of your transformation, it is a proper HTML document. What the XSLT engine reads in must be well-formed XML.

Assume that you have an XSL stylesheet that performs XML to HTML conversion. Everything works correctly with the exception of those HTML tags that end up as empty elements, that is, `<input type="text"/>`. For example, the following stylesheet creates an HTML document with an `<input>` element:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
...
<input type="text" name="{NAME}" size="{DISPLAY_LENGTH}" maxlength="{LENGTH}">
</input>
...
</xsl:stylesheet>
```

It renders HTML in the format of

```
<HTML>...<input type="text" name="in1" size="10" maxlength="20"/>
...
</HTML>
```

While Internet Explorer can handle this, Netscape cannot. Is there any way to generate completely cross-browser-compliant HTML with XSL?

The solution to this problem is that if you are seeing:

```
<input ... />
```

instead of:

```
<input ...></input>
```

then you are likely using the incorrect way of calling `XSLProcessor.processXSL()`, since it appears that it is not doing the HTML output for you. Use:

```
void processXSL(style,sourceDoc,PrintWriter)
```

instead of:

```
DocumentFragment processXSL(style,sourceDoc)
```

Creating an HTML Input Form

To generate an HTML form for inputting data using column names from the `user_tab_columns` table here is the XSL code:

```
<xsl:template match="ROW">
<xsl:value-of select="COLUMN_NAME"/>
 <INPUT NAME="{COLUMN_NAME}"/>
</xsl:template>
```

Correct XSL Namespace URI

The following URI is correct:

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

If you use:

```
xmlns:xsl="-- any other string here --"
```

it does not give correct output.

XSL Processor Produces Multiple Outputs

The XML Parser for Java, release 2.0.2.8 and above, supports `<ora:output>` to produce more than one result from one XML and XSL.

Keeping White Spaces in Your Output

Use this in your code, where (white spaces) means that you enter a space, newline, or tab there:

```
<xsl:text>...(white spaces)</xsl:text>
```

XDK Utilities That Translate Data from Other Formats to XML

XSLT translates from XML to XML, or to HTML, or to another text-based format. What about the other way around?

For HTML, you can use utilities like Tidy or JTidy to turn HTML into well-formed HTML that can be transformed using XSLT. For unstructured text formats, you can try utilities like XFlat at the following Web site:

`http://www.unidex.com/xflat.htm`

Multiple Threads Using a Single XSLProcessor and Stylesheet

Multiple threads can use a single XSLProcessor and XSLStylesheet instance to perform concurrent transformations. As long as you are processing multiple files with no more than one XSLProcessor and XSLStylesheet instance for each XML file you can do this simultaneously using threads.

Using Document Clones in Multiple Threads

It is safe to use clones of a document in multiple threads. The `public void setParam(String,String)` throws `XSLException` method of class `oracle.xml.parser.v2.XSLStylesheet` is supported. If you copy the global area set up by the constructor to another thread then it works. That method is supported since XML Parser for Java, release 2.0.2.5.

Disabling Output Escaping Is Supported

The XML Parser for Java provides an option to disable output escaping:

```
<xsl:text disable-output-escaping = "yes">
```

5

XML Schema Processor for Java

This chapter contains these topics:

What Is XML Schema?

XML Schema was created by the W3C to use XML itself to describe the content and the structure of XML documents. It includes most of the capabilities (it does not support `entity`) of Document Type Description (DTD) and additional capabilities.

See Also:

- `http://www.w3.org/TR/xmlschema-0/`

What Are DTDs?

A DTD is a mechanism provided by XML 1.0 for declaring constraints on XML markup. DTDs allow the specification of the following:

- Which elements or attributes can appear in your XML documents
- Which elements or attributes can be inside the elements
- The order the elements or attributes can appear

DTDs are also known as XML Markup Declarations.

XML Schema language serves a similar purpose to DTDs, but it is more flexible in specifying XML document constraints and potentially more useful for certain applications. Namespace support and datatypes support for elements and attributes are both found in XML Schema.

XML Schema is also referred to as XML Schema Definition (XSD).

DTD Limitations

DTDs are considered to be deficient in handling certain applications. DTD limitations include:

- DTD is not integrated with Namespace technology so users cannot import and reuse code
- DTD does not support datatypes other than character data, a limitation for describing metadata standards and database schemas

Applications need to specify document structure constraints more flexibly than the DTD can.

Comparison of XML Schema Features to DTD Features

Because of the inherent limitations of DTDs, the W3C is promoting XML Schema. XML Schema enables you to specify type information and constraints.

Table 5–1 lists XML Schema features compared to DTD features. Note that most XML Schema features include DTD features.

Table 5–1 XML Schema Features Compared to DTD Features

XML Schema Feature	DTD Feature
Built-In Datatypes	
XML schema specifies a set of built-in datatypes. Some of them are defined and called primitive datatypes, and they form the basis of the type system: `string, boolean, float, decimal, double, duration, dateTime, time, date, gYearMonth, gYear, gMonthDay, gMonth, gDay, Base64Binary, HexBinary, anyURI, NOTATION, QName.` Others are derived datatypes that are defined in terms of primitive types.	DTDs do not support datatypes other than character strings.
User-Defined Datatypes	

Users can derive their own datatypes from the built-in datatypes. There are three ways of datatype derivation: restriction, list and union.
Restriction defines a more restricted datatype by applying constraining facets to the base type
list simply allows a list of values of its item type
union defines a new type whose value can be of any of its member types
For example, to specify that the value of publish-year type to be within a specific range:
`<SimpleType name = "publish-year">`
`    <restriction base="gYear">`
`         <minInclusive value="1970"/>`
`         <maxInclusive value="2000"/>`
`    </restriction>`
`</SimpleType>`
The constraining facets are:
`length, minLength, maxLength, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, totalDigits, fractionDigits.`
Some facets only apply to certain base types.
	The publish-year element in the DTD example cannot be constrained further.

Table 5–1 (Cont.) XML Schema Features Compared to DTD Features

XML Schema Feature	DTD Feature
Occurrence Indicators (Content Model or Structure)	
In XML Schema, the structure (called `complexType`) of the instance document or an element is defined in terms of model group and attribute group. A model group may further contain model groups or element particles, while attribute group contains attributes. Wildcards can be used in both model group and attribute group to indicate any element or attribute. There are three kinds of model group: sequence, all, and choice, representing the sequence, conjunction and disjunction relationships among particles respectively. The range of the number of occurrence of each particle can also be specified. Like the datatype, `complexType` can be derived from other types. The derivation method can be either restriction or extension. The derived type inherits the content of the base type plus corresponding modifications. In addition to inheritance, a type definition can make references to other components. This feature allows a component to be defined once and used in many other structures. The type declaration and definition mechanism in XML Schema is much more flexible and powerful than the DTD.	Control by DTDs over the number of child elements in an element are assigned with the following symbols: ■ ? = zero or one. ■ * = zero or more ■ + = one or more ■ (none) = exactly one
Identity Constraints	-
XML Schema extends the concept of XML ID/IDREF mechanism with the declarations of unique, key and keyref. They are part of the type definition and allow not only attributes, but also element contents as keys. Each constraint has a scope within which it holds and the comparison is in terms of their value rather than lexical strings.	
Import/Export Mechanisms (Schema Import, Inclusion and Modification)	
All components of a schema need not be defined in a single schema file. XML Schema provides a mechanism of assembling multiple schemas. Import is used to integrate schemas of different namespace while inclusion is used to add components of the same namespace. Components can also be modified using redefinition when included.	You cannot use constructs defined in external schemas.

XML Schema Processor for Java Features

XML Schema Processor for Java, which is a part of the Oracle XDK Java components, has the following features:

- Fully supports the W3C XML Schema specifications of the Recommendation (May 2, 2001).
    - XML Schema Part 0: Primer
    - XML Schema Part 1: Structures
    - XML Schema Part 2: Datatypes
- Supports streaming (SAX) precessing, constant memory usage, and linear processing time.
- Built on the XML Parser for Java v2.

Supported Character Sets

XML Schema Processor for Java supports documents in the following encodings:

- BIG
- EBCDIC-CP-*
- EUC-JP
- EUC-KR
- GB2312
- ISO-2022-JP
- ISO-2022-KR
- ISO-8859-1to -9
- ISO-10646-UCS-2
- ISO-10646-UCS-4
- KOI8-R
- Shift_JIS
- US-ASCII
- UTF-8
- UTF-16

Requirements to Run XML Schema Processor for Java

To run XML Schema Processor for Java, you need the following:

- Any operating system with Java 1.2 support
- Java: JDK 1.2.x or higher.

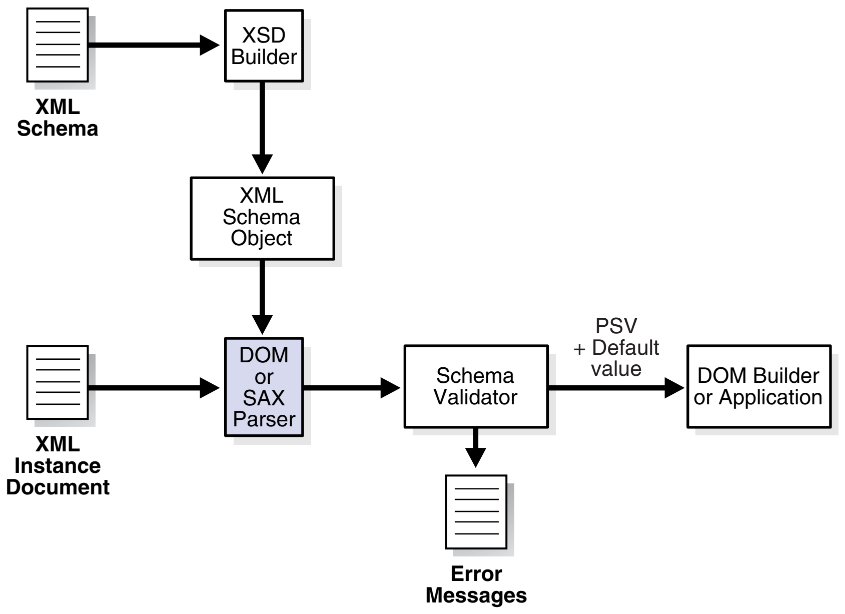
Documentation for sample programs for Oracle XML Schema Processor for Java is located in the file `xdk/demo/java/schema/README`.

XML Schema Processor for Java Usage

As shown in Figure 5–1, Oracle's XML Schema Processor for Java performs two major tasks:

- A builder (`XSDBuilder`) assembles schemas from XML Schema documents and passes XML Schema object to the DOM or SAX parser.
- A schema validator use the schemas to validate XML *instance* documents which have been read by the DOM or SAX parser.
- These results are passed on to a DOM builder or an application.
- Error messages are output by the schema validator.

Figure 5–1 XML Schema Processor for Java Usage



XML Schema can be used to define a class of XML documents. Instance document describes an XML document that conforms to a particular schema.

Although these instances and schemas need not exist specifically as "documents", they are commonly referred to as files. They may exist as any of the following:

- Streams of bytes
- Fields in a database record
- Collections of XML Infoset "Information Items"

When building the schema, the builder first compiles an internal schema object, and then calls the DOM Parser to parse the schema object into a corresponding DOM tree.

The validator works as a filter between the SAX Parser and your applications for the instance document. The validator takes SAX events of the instance document as input and validates them against the schema. If the validator detects invalid XML components it sends an error messages.

The output of the validator is:

- Input SAX events
- Default values it supplies
- Post-Schema Validation (PSV) information

Using the XML Schema API

The API of the XML Schema Processor for Java is simple. You can either use either of the following:

- `setSchemaValidationMode()` in the `DOMParser` as shown in `XSDSample.java`.
- Explicitly build the schema using `XSDBuilder` and set the schema for `XMLParser` as shown in `XSDSetSchema.java`.

If you do not explicitly set a compiled schema for validation using `XSDBuilder`, make sure that your instance document has the correct `xsi:schemaLocation` attribute pointing to the schema file. Otherwise, the validation will not be performed.

There is no clean-up call similar to `xmlclean`. If you need to release all memory and reset the state before validating a new XML document, terminate the context and start over.

XML Schema Processor for Java Sample Programs

The sample XML Schema Processor for Java files provided in the directory `/xdk/demo/java/schema` are described in Table 5–2:

Table 5–2 XML Schema Sample Files

File	Description
`cat.xsd`	The sample XML Schema definition file that supplies input to the `XSDSetSchema.java` program. XML Schema Processor for Java uses the XML Schema specification from `cat.xsd` to validate the contents of `catalogue.xml`.
`catalogue.xml`	The sample XML file that is validated by XML Schema processor against the XML Schema definition file, `cat.xsd`, using the program, `XSDSetSchema.java`.
`catalogue_e.xml`	When XML Schema Processor for Java processes this sample XML file using `XSDSample.java`, it generates XML Schema errors.
`DTD2Schema.java`	This sample program converts a DTD (first argument) into an XML Schema and uses it to validate an XML file (second argument).
`report.xml`	The sample XML file that is validated by XML Schema Processor for Java against the XML Schema definition file, `report.xsd`, using the program, `XSDSetSchema.java`.
`report.xsd`	The sample XML Schema definition file that is input to the `XSDSetSchema.java` program. XML Schema Processor for Java uses the XML Schema specification from `report.xsd` to validate the contents of `report.xml`.
`report_e.xml`	When XML Schema Processor for Java processes this sample XML file using `XSDSample.java`, it generates XML Schema errors.
`XSDSample.java`	Sample XML Schema Processor for Java program.
`XSDSetSchema.java`	When this example is run with `cat.xsd` and `catalogue.xml`, XML Schema Processor for Java uses the XML Schema specification from `cat.xsd` to validate the contents of `catalogue.xml`.
`XSDLax.java`	This example uses `SCHEMA_LAX_VALIDATION`.
`embeded_xsql.xsd`	The input file for `XSDLax.java.`
`embeded_xsql.xml`	The output file from `XSDLax.java.`

To run the sample programs:

1. Execute the program `make` to generate `.class` files.
2. Add `xmlparserv2.jar`, and the current directory to the `CLASSPATH`.

The following steps can be done in any order:

- Run the sample programs with the `XXX.xml` files:

```
java XSDSample report.xml
java XSDSetSchema report.xsd report.xml
java XSDLax embeded_xsql.xsd embeded_xsql.xml
```

XML Schema Processor for Java uses the XML Schema specification from `report.xsd` to validate the contents of `report.xml`.

- Run each sample program with the `catalogue.xml` file. For example:

```
java XSDSample   catalogue.xml
java XSDSetSchema cat.xsd  catalogue.xml
```

XML Schema Processor for Java uses the XML Schema specification from `cat.xsd` to validate the contents of catalogue.xml.

- The following are samples that find XML Schema errors:

```
java XSDSample catalogue_e.xml
java XSDSample report_e.xml
```

- Run the sample for converting a DTD to an XML Schema.

```
java DTD2Schema dtd2schema.dtd dtd2schema.xml
```

6 Using JAXB Class Generator

This chapter contains these topics:

What Is JAXB?

Java Architecture for XML Binding (**JAXB**) consists of an API and tools that map to and from XML data and Java objects. It is an implementation of the JSR-31 "The Java Architecture for XML Binding (JAXB)", Version 1.0, recommendation of the JCP (Java Community Process). **JSR** is a Java Specification Request of the JCP.

The JAXB compiler generates the interfaces and the implementation classes corresponding to the XML Schema. The classes can be used to read, manipulate and re-create XML documents. The JAXB compiler generates Java classes corresponding to an XML Schema, and interfaces that are needed to access XML data. The Java classes, which can be extended, give you access to the XML data without any specific knowledge about the underlying data structure.

Note:

- The JAXB specification is described at `http://java.sun.com/xml/jaxb/`
- JSR is described at `http://jcp.org/en/jsr/overview`

Replacing the XML Class Generator with JAXB Class Generator

You are requested to use JAXB Class Generator for new applications in order to use the object binding feature for XML data. The Oracle9*i* Class Generator for Java is deprecated. However, the Oracle9*i* Class Generator runtime was included in release 10.1 and is supported for the duration of the 10.*x* releases.

Unmarshalling, Marshalling, and Validating Using JAXB

Unmarshalling is defined as moving data from an XML document to the Java generated class objects. Each object is derived from an instance of the schema component in the input document. Because of the inherent weaknesses of DTDs, they are not supported by JAXB, but a DTD can be converted to an XML Schema that is then used by JAXB.

Marshalling is defined as creating an XML document from Java objects by traversing a content tree of instances of Java classes.

Validation is a prerequisite to marshalling if content has changed in the Java representation. Validation is verifying that the content tree satisfies the constraints defined in the schema. The tree is defined as valid when marshalling the tree generates a document that is valid according to the source schema. If unmarshalling includes validation that is error-free then the input document and the content tree are valid. However, validation is not required during unmarshalling.

Validation comes in these forms:

- Unmarshalling-time validation notifies the application of errors and warnings during unmarshalling.
- On-demand validation on a Java content tree when the application initiates it.
- Fail-fast validation that gives immediate results.

Using JAXB Class Generator

To build a JAXB application, start with an XML Schema file. Build and use a JAXB application by performing these steps:

1. Generate the Java source files by submitting the XML Schema file to the binding compiler. The binding compiler could be invoked through the command-line utility called `orajaxb`.
2. Compile the Java source code using JDK 1.3 or higher.
3. With the classes and the binding framework, write Java applications that:
   - Build object trees representing XML data that is valid against the XML Schema by either unmarshalling the data from a document or instantiating the classes you created.
   - 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` for more information on JAXB

Features Not Supported in JAXB

The Oracle release does not support the following:

- The Javadoc generation.
- The `List` and `Union` features of XML Schema.
- `SimpleType` mapping to `TypeSafe Enum` class and `IsSet` property modifier.

- XML Schema component "any" and substitution groups.
- Customization of XML Schema to override the default binding of XML Schema components.
- On-demand validation of content tree.

JAXB Class Generator Command-Line Interface

The JAXB class generator command-line interface is accessed this way:

```
oracle.xml.jaxb.orajaxb [-options]
```

where the options are listed in Table 6–1:

Table 6–1 JAXB Class Generator Command-line Interface

Option	Description
`-help`	Prints the help message text
`-version`	Prints the release version
`-outputDir` *`OutputDir`*	The directory in which to generate Java source
`-schema` *`SchemaFile`*	The input schema file
`-targetPkg` *`targetPkg`*	The target package name
`-interface`	Generate only the interfaces

JAXB Compared with JAXP

The following lists summarize the advantages of JAXB.

Use JAXB when you want to:

- Access data in memory, but do not need DOM tree manipulation capabilities.
- Build object representations of XML data.

For more information about JAXB:

See Also:

- Code examples are found with the XDK download and with the JAXB code at `$INSTALL_HOME/xdk/demo/java/jaxb`
- `http://java.sun.com/xml/jaxb/` for documentation and examples of the use of JAXB
- *Oracle XML API Reference* for details of the JAXB API

The Java API for XML Processing (JAXP), that enables applications to parse and transform XML documents using an API that is independent of a particular XML processor implementation, is implemented by Oracle.

See Also: "Using JAXP" on page 3-36

Use JAXP when you want to:

- Have flexibility with regard to the way you access the data: either serially with SAX or randomly in memory with DOM.
- Use your same processing code with documents based on different schemas.

- Parse documents that are not necessarily valid.
- Apply XSLT transforms.
- Insert or remove objects from an object tree that represents XML data.

7 XML SQL Utility (XSU)

This chapter contains these topics:

What Is XML SQL Utility (XSU)?

XML has become the format for data interchange, but at the same time, a substantial amount of data resides in object-relational databases. It is therefore necessary to have the ability to transform this object-relational data to XML.

XML SQL Utility (XSU) enables you to do these transformations:

- XSU can transform data retrieved from object-relational database tables or views into XML.
- XSU can extract data from an XML document, and using a given mapping, insert the data into appropriate columns or attributes of a table or a view.
- XSU can extract data from an XML document and apply this data to updating or deleting values of the appropriate columns or attributes.

Generating XML from the Database

When given a `SELECT` query, XSU queries the database and returns the results as an XML document.

Storing XML in the Database

Given an XML document, XSU can extract the data from the document and insert it into a table in the database.

Accessing XSU Functionality

XML SQL Utility functionality can be accessed in the following ways:

- Through a Java API
- Through a PL/SQL API
- Through a Java command-line front end

XSU Features

- Dynamically generates DTDs.
- During generation, performs simple transformations, such as modifying default tag names for the ROW element. You can also register an XSL transformation that is then applied to the generated XML documents as needed.
- Generates XML documents in their string or DOM representations.
- Inserts XML into database tables or views. XSU can also update or delete records from a database object, given an XML document.
- Generates complex nested XML documents. XSU can also store them in relational tables by creating object views over the flat tables and querying over these views. Object views can create structured data from existing relational data using object-relational infrastructure.
- Generates an XML Schema given a SQL query.
- Generates XML as a stream of SAX2 callbacks.
- Supports XML attributes during generation. This provides an easy way to specify that a particular column or group of columns must be mapped to an XML attribute instead of an XML element.
- Allows SQL identifier to XML identifier escaping. Sometimes column names are not valid XML tag names. To avoid this you can either alias all the column names or turn on tag escaping.
- Supports `XMLType` columns in objects or tables.

See Also:

- *Oracle XML DB Developer's Guide*, in particular, the chapter on generating XML, for examples on using XSU with XMLType
- *Oracle XML API Reference* for more information on the Java API
- *PL/SQL Packages and Types Reference*
- Chapter 24, "XSU for PL/SQL"

XSU Dependencies and Installation

Important information about XSU:

Note: In Oracle9*i*, `XMLGen` was deprecated and is now no longer included with Oracle software. The replacements for `XMLGEN` are the packages `DBMS_XMLQuery`, used for XML generation, and `DBMS_XMLSave`, used for DML and data manipulation.

Migration is simple: the method names are identical. The new XSU for PL/SQL now contains more methods. All methods take the context handle as the first argument.

Dependencies of XSU

XML SQL Utility (XSU) depends on the following components:

- Database connectivity - JDBC drivers. XSU can work with any JDBC driver but it is optimized for Oracle JDBC drivers. Oracle does not make any guarantee or provide support for the XSU running against non-Oracle databases.
- Oracle XML Parser, Version2 `- xmlparserv2.jar`. This file is included in the Oracle installations. `xmlparserv2.jar` is also part of the XDK Java components archive downloadable from Oracle Technology Network (OTN) Web site.
- XSU also depends on the classes included in `xdb.jar` and `servlet.jar`. These are present in Oracle installations. These are also included in the XDK Java components archive downloadable from OTN.

Installing XSU

XSU is on the Oracle software CD, and it is also part of the XDK Java components package available on OTN. The XSU comes in the form of two files:

- `$ORACLE_HOME/lib/xsu12.jar` -- Contains all the Java classes that make up XSU. `xsu12.jar` requires a minimum of JDK1.2 and JDBC2
- `$ORACLE_HOME/rdbms/admin/dbmsxsu.sql` -- This is the SQL script that builds the XSU PL/SQL API. Load `xsu12.jar` into the database before `dbmsxsu.sql` is executed.

By default, the Oracle installer installs the XSU on the hard drive in the locations specified in the previous bulleted paragraphs. It also loads the XSU into the database.

If XSU is not installed during the initial Oracle installation, it can be installed later. You can either use Oracle Installer to install the XSU and its dependent components, or you can download the latest XDK Java components from OTN.

To load the XSU into the database you need to take one of the following steps, depending on how you installed XSU:

- Oracle Installer installation: Change directory to your ORACLE_HOME directory, then to `rdbms/admin`. Run `initxml.sql`.
- OTN download installation: Change directory into the `bin` directory of the downloaded and expanded XDK tree. Then run script `xdk load`. Windows users run `xdkload.bat`.

Where XSU can be Installed

XSU is written in Java, and can live in any tier that supports Java. XSU can be installed on a client system.

XML SQL Utility in the Database

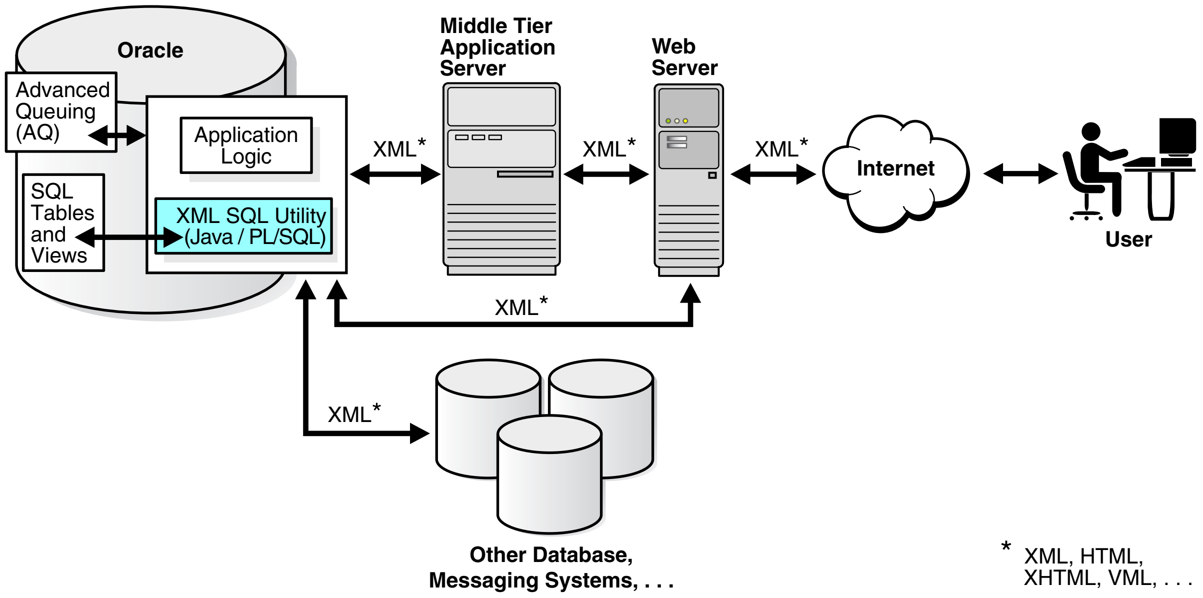
The Java classes that make up XSU can be loaded into a Java-enabled Oracle database. XSU contains a PL/SQL wrapper that publishes the XSU Java API to PL/SQL, creating a PL/SQL API. This way you can:

- Write new Java applications that run inside the database and that can directly access the XSU Java API
- Write PL/SQL applications that access XSU through its PL/SQL API
- Access XSU functionality directly through SQL

Figure 7–1 shows the typical architecture for such a system. 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. The XML can be used from within stored procedures in the database or shipped outside through web servers or application servers.

In Figure 7–1, all lines are bi-directional. Since XSU can *generate* as well as *save* data, data can come from various sources to XSU running inside the database, and can be put back in the appropriate database tables.

Figure 7–1 Running XML SQL Utility in the Database


Oracle
Advanced Queuing (AQ)
SQL Tables and Views
Application Logic
XML SQL Utility (Java / PL/SQL)
Middle Tier Application Server
Web Server
XML*
XML*
XML*
Internet
User
XML*
XML*
Other Database, Messaging Systems, . . .
* XML, HTML, XHTML, VML, . . .

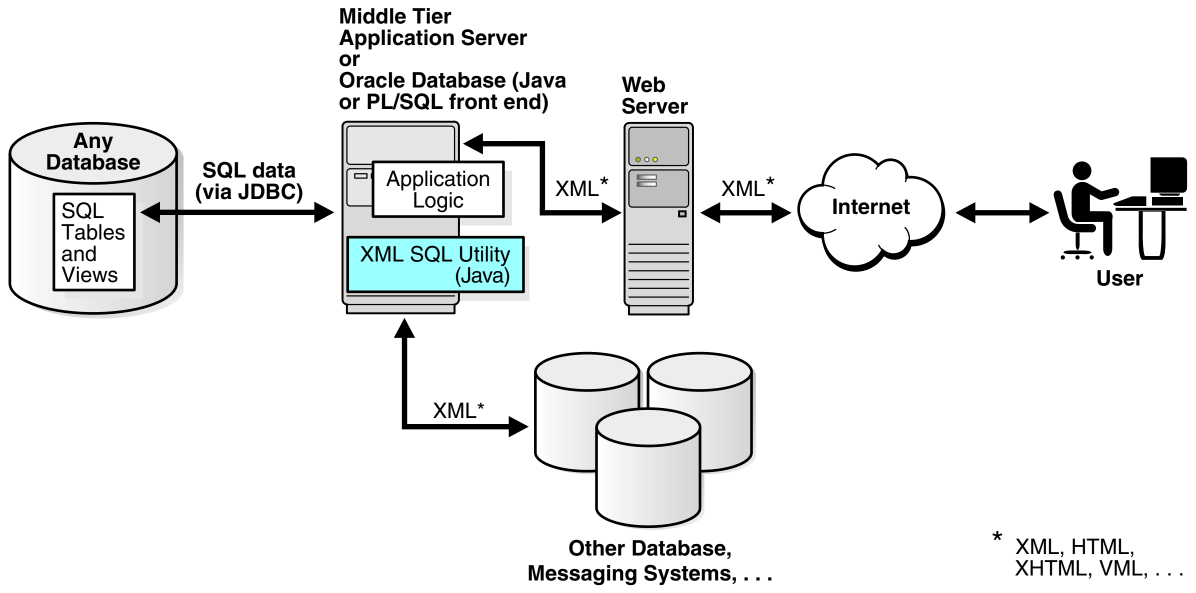
XML SQL Utility in the Middle Tier

Your application architecture may need to use an application server in the middle tier, separate from the database. 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 and your Java programs can make use of XSU through its Java API.

Figure 7–2, shows how 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 whole process is bi-directional and the data can be put back into the JDBC sources (database tables or views) using XSU. If an Oracle database itself is used as the application server, then you can also use the PL/SQL front-end instead of Java.

Figure 7–2 Running XML SQL Utility in the MIddle Tier



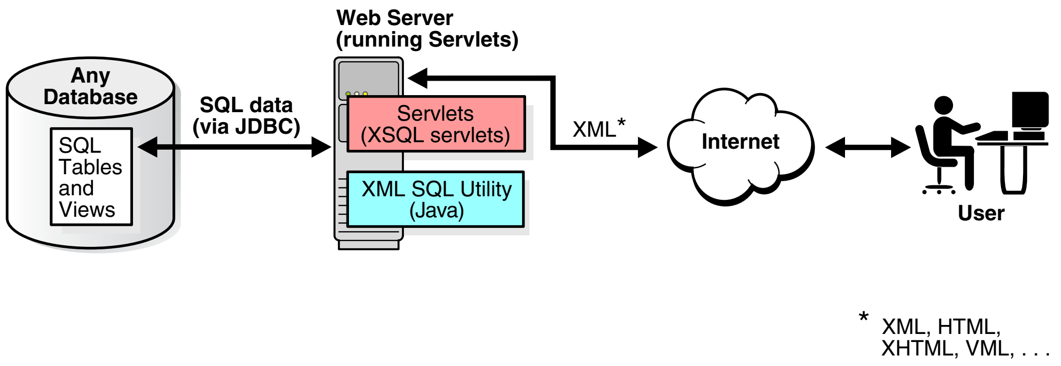
XML SQL Utility in a Web Server

Figure 7–3 XSU can live in the Web server, as long as the Web server supports Java servlets. This way you can write Java servlets that use XSU to accomplish their task.

XSQL Servlet does just this. 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 do XML processing in the Web server, you can use the XSQL Servlet, because it spares you from the intricate servlet programming.

> **See:** Chapter 8, "XSQL Pages Publishing Framework" for information about using XSQL Servlet.

Figure 7–3 Running XML SQL Utility in a Web Server



SQL-to-XML and XML-to-SQL Mapping Primer

This section describes the mapping or transformation used to go from SQL to XML or vice versa.

Default SQL-to-XML Mapping

Consider table `emp1`:

```
CREATE TABLE emp1
(
   empno NUMBER,
   ename VARCHAR2(20),
   job VARCHAR2(20),
   mgr  NUMBER,
   hiredate DATE,
   sal NUMBER,
   deptno NUMBER
);
```

XSU can generate an XML document by specifying the query:

```
select * from emp1:

<?xml version='1.0'?>
<ROWSET>
  <ROW num="1">
    <EMPNO>7369</EMPNO>
    <ENAME>sMITH</ENAME>
    <JOB>clerk</JOB>
    <mgr>7902</mgr>
    <HIREDATE>12/17/1980 0:0:0</HIREDATE>
    <SAL>800</SAL>
    <DEPTNO>20</DEPTNO>
  </ROW>
  <!-- additional rows ... -->
</ROWSET>
```

In the generated XML, the rows returned by the SQL query are enclosed in a `ROWSET` tag to constitute the `<ROWSET>` element. This element is also the root element of the generated XML document.

- The `<ROWSET>` element contains one or more `<ROW>` elements.

- Each of the `<ROW>` elements contain the data from one of the returned database table rows. Specifically, each `<ROW>` element contains one or more elements whose names and content are those of the database columns specified in the `SELECT` list of the SQL query.
- These elements, corresponding to database columns, contain the data from the columns.

SQL-to-XML Mapping Against Object-Relational Schema

Here is a mapping against an object-relational schema: Consider the object type, `AddressType`. It is an object type whose attributes are all scalar types and is created as follows:

```
CREATE TYPE AddressType AS OBJECT (
   street VARCHAR2(40),
   city   VARCHAR2(20),
   state  CHAR(2),
   zip    VARCHAR2(10)
);
```

The following type, `EmployeeType`, is also an object type but it has an `empaddr` attribute that is of an object type itself, specifically, `AddressType`. `Employee Type` is created as follows:

```
CREATE TYPE EmployeeType AS OBJECT
(
  empno NUMBER,
  ename VARCHAR2(20),
  salary NUMBER,
  empaddr AddressType
);
```

The following type, `EmployeeListType`, is a collection type whose elements are of the object type, `EmployeeType`. `EmployeeListType` is created as follows:

```
CREATE TYPE EmployeeListType AS TABLE OF EmployeeType;
```

Finally, `dept1` is a table with an object type column and a collection type column: `AddressType` and `EmployeeListType` respectively.

```
CREATE TABLE dept1
(
  deptno NUMBER,
  deptname VARCHAR2(20),
  deptaddr AddressType,
  emplist  EmployeeListType
)
NESTED TABLE emplist STORE AS emplist_table;
```

Assume that valid values are stored in table, `dept1`. For the query `select * from dept1`, XSU generates the following XML document:

```
<?xml version='1.0'?>
<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>
    <EMPLIST>
      <EMPLIST_ITEM num="1">
         <EMPNO>7369</EMPNO>
         <ENAME>John</ENAME>
         <SALARY>10000</SALARY>
         <EMPADDR>
           <STREET>300 Embarcadero</STREET>
           <CITY>Palo Alto</CITY>
           <STATE>CA</STATE>
           <ZIP>94056</ZIP>
         </EMPADDR>
      </EMPLIST_ITEM>
       <!-- additional employee types within the employee list -->
    </EMPLIST>
  </ROW>
  <!-- additional rows ... -->
</ROWSET>
```

As in the last example, the mapping is canonical, that is, `<ROWSET>` contains `<ROW>` elements that contain elements corresponding to the columns. As before, the elements corresponding to scalar type columns simply contain the data from the column.

Mapping Complex Type Columns to XML

Things get more complex with elements corresponding to a complex type column. For example, `<DEPTADDR>` corresponds to the `DEPTADDR` column which is of object type `ADDRESS`. Consequently, `<DEPTADDR>` contains sub-elements corresponding to the attributes specified in the type `ADDRESS`. These sub-elements can contain data or sub-elements of their own, again depending if the attribute they correspond to is of a simple or complex type.

Mapping Collections to XML

When dealing with elements corresponding to database collections, things are also different. Specifically, the `<EMPLIST>` element corresponds to the `EMPLIST` column which is of a `EmployeeListType` collection type. Consequently, the `<EMPLIST>` element contains a list of `<EMPLIST_ITEM>` elements, each corresponding to one of the elements of the collection.

Other observations to make about the preceding mapping are:

- 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 particular column is mapped to an XML attribute instead of an XML element.

Customizing the Generated XML: Mapping SQL to XML

Often, you need to generate XML with a specific structure. Since 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 using one of the following methods:

- "Source Customizations"
- "Mapping Customizations"

- "Post-Generation Customizations"

Source Customizations

Source customizations are done by altering the query or 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:
  - Use cursor subqueries, or cast-multiset constructs to get nesting in the XML document that comes from a flat schema.
  - Alias column and attribute names to get the desired XML element names.
  - Alias top level scalar type columns with identifiers that begin with the at sign (@) to have them map to an XML attribute instead of an XML element. For example, `SELECT empno AS "@empno",... FROM emp`, results in an XML document where the `<ROW>` element has an attribute `EMPNO`.

Mapping Customizations

XML SQL Utility enables you to modify the mapping it uses to transform SQL data into XML. You can make any of the following SQL to XML mapping changes:

- Change or omit the `<ROWSET>` tag.
- Change or omit the `<ROW>` tag.
- Change or omit the attribute `num`. This 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 using a nullness attribute, rather then by omission of the element.

Post-Generation Customizations

Finally, if the desired customizations cannot be achieved with the foregoing methods, you can write an XSL transformation and register it with XSU. While there is an XSLT registered with the XSU, XSU can apply the XSLT to any XML it generates.

Default XML-to-SQL Mapping

XML to SQL mapping is just the reverse of the SQL to XML mapping.

Consider the following differences when mapping from XML to SQL, compared to mapping from SQL to XML:

- When going from XML to SQL, the XML attributes are ignored. Thus, there is really no mapping of XML attributes to SQL.
- When going from SQL to XML, mapping is performed from the `ResultSet` created by the SQL query to XML. This way the query can span multiple database tables or views. What is formed is a single `ResultSet` that is then converted into XML. This is not the case when going from XML to SQL, where:

  - 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, one solution is to use `INSTEAD-OF-INSERT` triggers.

If the XML document does not perfectly map into the target database schema, there are three things you can do:

- Modify the Target. Create an object-relational view over the target schema, and make the view the new target.
- Modify the XML Document. Use XSLT to transform the XML document. The XSLT can be registered with XSU so that the incoming XML is automatically transformed, before any mapping attempts are made.
- Modify XSU's XML-to-SQL Mapping. You can instruct XSU to perform case insensitive matching of the XML elements to database columns or attributes.
  - You can tell XSU to use the name of the element corresponding to a database row instead of `ROW`.
  - You can specify in XSU the date format to use when parsing dates in the XML document.

How XML SQL Utility Works

This section describes how XSU works when performing the following tasks:

- Selecting with XSU on page 7-10
- Queries That XSU Cannot Handle
- Inserting with XSU on page 7-10
- Updating with XSU on page 7-11
- Deleting with XSU on page 7-12

Selecting with XSU

XSU generation is simple. SQL queries are executed and the `ResultSet` is retrieved from the database. Metadata about the ResultSet is acquired and analyzed. Using the mapping described in "Default SQL-to-XML Mapping" on page 7-6, the SQL result set is processed and converted into an XML document.

Queries That XSU Cannot Handle

There are certain types of queries that XSU cannot handle, especially those that mix columns of type `LONG` or `LONG RAW` with `CURSOR()` expressions in the Select clause. Please note that `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 `CLOB`s, then the queries will succeed.

Inserting with XSU

To insert the contents of an XML document into a particular table or view, XSU first retrieves the metadata about the target table or view. Based on the metadata, XSU generates a SQL `INSERT` statement. XSU extracts the data out of the XML document and binds it to the appropriate columns or attributes. Finally the statement is executed.

For example, assume that the target table is `dept1` and the XML document is the one generated from `dept1`.

XSU generates the following `INSERT` statement.

```
INSERT INTO dept1 (deptno, deptname, deptaddr, emplist) VALUES (?,?,?,?)
```

Next, the XSU parses the XML document, and for each record, it binds the appropriate values to the appropriate columns or attributes:

```
deptno   <- 100
deptname <- SPORTS
deptaddr <- AddressType('100 Redwood Shores Pkwy','Redwood Shores',
                        'CA','94065')
emplist  <- EmployeeListType(EmployeeType(7369,'John',100000,
            AddressType('300 Embarcadero','Palo Alto','CA','94056'),...)
```

The statement is then executed. Insert processing can be optimized to insert in batches, and commit in batches.

See Also:

- "Default SQL-to-XML Mapping" on page 7-6
- "Insert Processing Using XSU (Java API)" on page 7-26 for ore detail on batching

Updating with XSU

Updates and deletes 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 there are no triggers or constraints on the table.

However, with both updates and deletes, the XML element can match more than one row if the matching columns are not key columns in the table. For updates, you must provide a list of key columns that XSU needs to identify the row to update. For example, to update the `DEPTNAME` to `SportsDept` instead of `Sports`, you can have an XML document such as:

```
<ROWSET>
  <ROW num="1">
    <DEPTNO>100</DEPTNO>
    <DEPTNAME>SportsDept</DEPTNAME>
  </ROW>
</ROWSET>
```

and supply the `DEPTNO` as the key column. This results in the following `UPDATE` statement:

```
UPDATE dept1 SET deptname = ? WHERE deptno = ?
```

and bind the values this way:

```
deptno <- 100
deptname <- 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: "Update Processing Using XSU (Java API)" on page 7-28

Deleting with XSU

For deletes, you can choose to give a set of key columns for the delete to identify the rows. If the set of key columns are not given, then the `DELETE` statement tries to match all the columns given in the document. For an XML document:

```
<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>
```

To delete, XSU builds a `DELETE` statement (one for each `ROW` element):

```
DELETE FROM dept1 WHERE deptno = ? AND deptname = ? AND deptaddr = ?
```

The binding is:

```
deptno   <- 100
deptname <- sports
deptaddr <- addresstype('100 redwood shores pkwy','redwood city','ca',
            '94065')
```

See Also: "Delete Processing Using XSU (Java API)" on page 7-30

Using the XSU Command-Line Front End OracleXML

XSU comes with a simple command line front end that gives you quick access to XML generation and insertion.

The XSU command-line options are provided through the Java class, `OracleXML`. Invoke it by calling:

```
java OracleXML
```

This prints the front end usage information. To run the XSU command-line front end, first specify where the executable is located. Add the following to your `CLASSPATH`:

- XSU Java library (`xsu12.jar` or `xsu111.jar`)

Also, since XSU depends on Oracle XML Parser and JDBC drivers, make the location of these components known. To do this, the `CLASSPATH` must include the locations of:

- Oracle XML Parser Java library (`xmlparserv2.jar`)
- JDBC library (`classes12.jar` if using `xsu12.jar` or `classes111.jar` if using `xsu111.jar`)
- A JAR file for `XMLType`.

Generating XML Using the XSU Command Line

For XSU generation capabilities, use the XSU `getXML` parameter. For example, to generate an XML document by querying the `employees` table in the `hr` schema, use:

```
java OracleXML getXML -user "hr/hr" "select * from employees"
```

This performs the following tasks:

1. Connects to the current default database
2. Executes the query `select * from employees`
3. Converts the result to XML
4. Displays the result

The `getXML` parameter supports a wide range of options. They are explained in the following section.

XSU's OracleXML getXML Options

Table 7–1 lists the OracleXML `getXML` options:

Table 7–1 XSU's OracleXML getXML Options

getXML Option	Description
`-user` *`username/password`*	Specifies the username and password to connect to the database. If this is not specified, the user defaults to `scott/tiger`. Note that the connect string is also being specified. The username and password can be specified as part of the connect string.
`-conn` *`JDBC_connect_string`*	Specifies the JDBC database connect string. By default the connect string is: "`jdbc:oracle:oci:@`"):
`-withDTD`	Instructs the XSU to generate the DTD along with the XML document.
`-withSchema`	Instructs the XSU to generate the schema along with the XML document.
`-rowsetTag` *`tag_name`*	Specifies `rowset` tag (the tag that encloses all the XML elements corresponding to the records returned by the query). The default `rowset` tag is `ROWSET`. Specifying an empty string for the `rowset` tells the XSU to completely omit the `rowset` element.
`-rowTag` *`tag_name`*	Specifies the `row` tag (the tag used to enclose the data corresponding to a database row). The default row tag is `ROW`. Specifying an empty string for the row tag tells the XSU to completely omit the row tag.
`-rowIdAttr` *`row_id_attribute_name`*	Names the attribute of the `ROW` element keeping track of the cardinality of the `rows`. By default this attribute is called `num`. Specifying an empty string ("") as the `rowID` attribute will tell the XSU to omit the attribute.
-rowIdColumn *`row_Id_column_name`*	Specifies that the value of one of the scalar columns from the query is to be used as the value of the `rowID` attribute.
`-collectionIdAttr` *`collection_id_ attribute name`*	Names the attribute of an XML list element keeping track of the cardinality of the elements of the list (the generated XML lists correspond to either a cursor query, or collection). Specifying an empty string ("") as the `rowID` attribute will tell the XSU to omit the attribute.
`-useNullAttrId`	Tells the XSU to use the attribute `NULL (TRUE/FALSE)` to indicate the nullness of an element.
`-styleSheet` *`stylesheet_URI`*	Specifies the stylesheet in the XML PI (Processing Instruction).

Table 7–1 (Cont.) XSU's OracleXML getXML Options

getXML Option	Description
`-stylesheetType` *`stylesheet_type`*	Specifies the stylesheet type in the XML PI (Processing Instruction).
`-errorTag` *`error tag_name`*	Specifies the error tag - the tag to enclose error messages that are formatted into XML.
`-raiseNoRowsException`	Tells the XSU to raise an exception if no rows are returned.
`-maxRows` *`maximum_rows`*	Specifies the maximum number of rows to be retrieved and converted to XML.
`-skipRows` *`number_of_rows_to_skip`*	Specifies the number of rows to be skipped.
`-encoding` *`encoding_name`*	Specifies the character set encoding of the generated XML.
`-dateFormat` *`date_format`*	Specifies the date format for the date values in the XML document.
`-fileName` *`SQL_query_fileName`* | *`sql_query`*	Specifies the file name that contains the query, or specify the query itself.
`-useTypeForCollElemTag`	Use type name for column-element tag (by default XSU uses the `column-name_item`.
`-setXSLTRef` *`URI`*	Set the XSLT external entity reference.
`-useLowerCase` | `-useUpperCase`	Generate lowercase or uppercase tag names, respectively. The default is to match the case of the SQL object names from which the tags are generated.
`-withEscaping`	There are characters that are legal in SQL object names but illegal in XML tags. This option means that if such a character is encountered, it is escaped rather than throwing an exception.
`-raiseException`	By default the XSU catches any error and produces the XML error. This changes this behavior so the XSU actually throws the raised Java exception.

Inserting XML Using XSU's Command Line (putXML)

To insert an XML document into the `employees` table in the `hr` schema, use the following syntax:

```
java OracleXML putXML -user "hr/hr" -fileName "/tmp/temp.xml" "employees"
```

This performs the following tasks:

1. Connects to the current database
2. Reads the XML document from the given file
3. Parses it, matches the tags with column names
4. Inserts the values appropriately into the `employees` table

Note: The XSU command line front end, `putXML`, currently only publishes XSU insert functionality.

XSU OracleXML putXML Options

Table 7–2 lists the `putXML` options:

Table 7–2 XSU's OracleXML putXML Options

putXML Options	Description
`-user` *`username/password`*	Specifies the username and password to connect to the database. If this is not specified, the user defaults to `scott/tiger`. The connect string is also being specified; the username and password can be specified as part of the connect string.
`-conn` *`JDBC_connect_string`*	Specifies the JDBC database connect string. By default the connect string is: "`jdbc:oracle:oci:@`"):
`-batchSize` *`batching_size`*	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.
`-commitBatch` *`commit_size`*	Specifies the number of inserted records after which a commit is to be executed. Note that if the autocommit is `TRUE` (the default), then setting the `commitBatch` has no consequence.
`-rowTag` *`tag_name`*	Specifies the `row` tag (the tag used to enclose the data corresponding to a database row). The default row tag is `ROW`. Specifying an empty string for the `row` tag tells XSU that no row-enclosing tag is used in the XML document.
`-dateFormat` *`date_format`*	Specifies the date format for the date values in the XML document.
`-ignoreCase`	Makes the matching of the column names with tag names case insensitive (for example, "`EmpNo`" will match with "`EMPNO`" if ignoreCase is on).
`-fileName` *`file_name`*	Specifies the XML document to insert, a local file.
`-URL` *`URL`*	Specifies a URL to fetch the document from.
`-xmlDoc` *`xml_document`*	Specifies the XML document as a string on the command line.
`-tableName` *`table`*	The name of the table to put the values into.
`-withEscaping`	If SQL to XML name escaping was used when generating the doc, then this will turn on the reverse mapping.
`-setXSLT` *`URI`*	XSLT to apply to the XML document before inserting.
`-setXSLTRef` *`URI`*	Set the XSLT external entity reference.

XSU Java API

The following two classes make up the XML SQL Utility Java API:

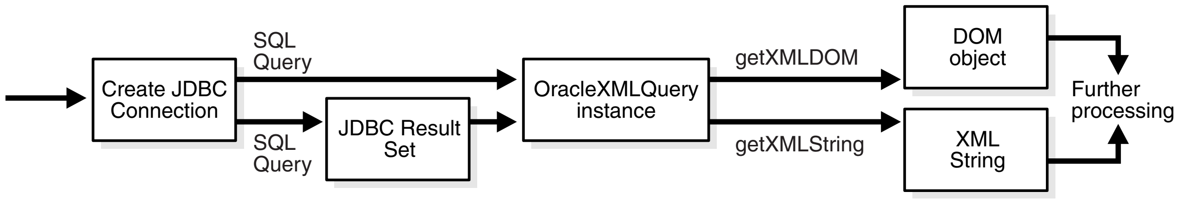
- XSU API for XML generation: `oracle.xml.sql.query.OracleXMLQuery`
- XSU API for XML save, insert, update, and delete: `oracle.xml.sql.dml.OracleXMLSave`

Generating XML with XSU's OracleXMLQuery

The `OracleXMLQuery` class makes up the XML generation part of the XSU Java API. Figure 7–4 illustrates the basic steps you need to take when using OracleXMLQuery to generate XML:

1. Create a connection.
2. Create an `OracleXMLQuery` instance by supplying an SQL string or a `ResultSet` object.
3. Obtain the result as a DOM tree or XML string.

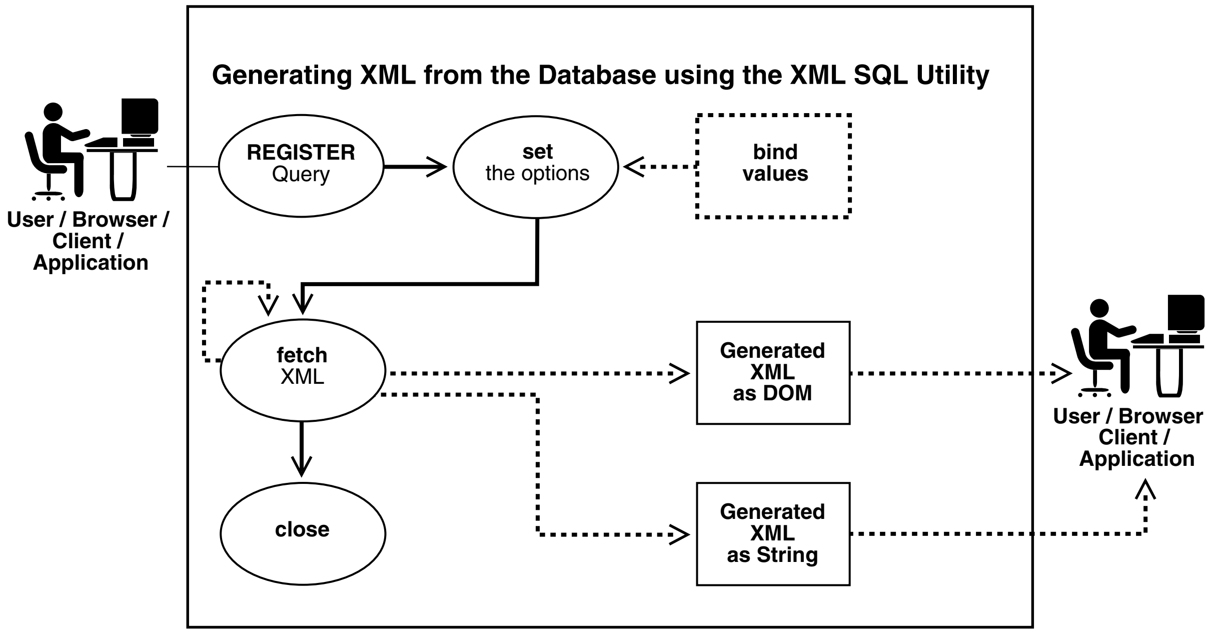
Figure 7–4   Generating XML With XML SQL Utility for Java: Basic Steps



Generating XML from SQL Queries Using XSU

The following examples illustrate how XSU can generate an XML document in its DOM or string representation given a SQL query. See Figure 7–5.

Figure 7–5   Generating XML With XML SQL Utility


Generating XML from the Database using the XML SQL Utility
User / Browser / Client / Application
REGISTER Query
set the options
bind values
fetch XML
Generated XML as DOM
Generated XML as String
close
User / Browser Client / Application

XSU Generating XML Example 1: Generating a String from Table employees (Java)

1. Create a connection

- Before generating the XML you must create a connection to the database. The connection can be obtained by supplying the JDBC connect string. First register the Oracle JDBC class and then create the connection, as follows

```
// import the Oracle driver..
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","hr");
```

Here, we use the default connection for the JDBC OCI driver. You can connect to the `scott` schema supplying the password `tiger`.

You can also use the JDBC thin driver to connect to the database. The thin driver is written in pure Java and can be called from within applets or any other Java program.

See Also:: *Oracle Database Java Developer's Guide* for more details.

- Here is an example of connecting using the JDBC thin driver:

```
// Create the connection.
Connection conn =
   DriverManager.getConnection("jdbc:oracle:thin:@dlsun489:1521:ORCL",
   "hr","hr");
```

The thin driver requires you to specify the host name (`dlsun489`), port number (1521), and the Oracle SID (`ORCL`), which identifies a specific Oracle instance on the machine.

- No connection is needed when run in the server. When writing server side Java code, that is, when writing code that will run in the server, you need not establish a connection using a username and password, since the server-side internal driver runs within a default session. You are already connected. In this case call the `defaultConnection()` on the `oracle.jdbc.driver.OracleDriver()` class to get the current connection, as follows:

```
import oracle.jdbc.*;
// Load the Oracle JDBC driver
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
Connection conn =  new oracle.jdbc.OracleDriver().defaultConnection ();
```

The remaining discussion either assumes you are using an OCI connection from the client or that you already have a connection object created. Use the appropriate connection creation based on your needs.

Note:
`oracle.xml.sql.dataset.OracleXMLDataSetExtJdbc` is used only for Oracle JDBC, while `oracle.xml.sql.dataset.OracleXMLDataSetGenJdbc` is used for non-Oracle JDBC.

2. Creating an `OracleXMLQuery` Class instance:

Once you have registered your connection, create an `OracleXMLQuery` class instance by supplying a SQL query to execute as follows:

```
// import the query class in to your class
import oracle.xml.sql.query.OracleXMLQuery;

OracleXMLQuery qry = new OracleXMLQuery (conn, "select * from employees");
```

You are now ready to use the query class.

3. Obtain the result as a DOM tree or XML string:

- DOM object output. If, instead of a string, you wanted a DOM object, you can simply request a DOM output as follows:

  ```
  org.w3c.DOM.Document domDoc = qry.getXMLDOM();
  ```

  and use the DOM traversals.

- XML string output. You can get an XML string for the result by:

  ```
  String xmlString = qry.getXMLString();
  ```

Here is a complete listing of the program to extract (generate) the XML string. This program gets the string and prints it out to standard output:

```
import oracle.jdbc.*;
import oracle.xml.sql.query.OracleXMLQuery;
import java.lang.*;
import java.sql.*;

// class to test the String generation!
class testXMLSQL {

   public static void main(String[] argv)
   {

     try{
      // create the connection
      Connection conn  = getConnection("hr","hr");

      // Create the query class.
      OracleXMLQuery qry = new OracleXMLQuery(conn, "select * from employees");

      // Get the XML string
      String str = qry.getXMLString();

      // Print the XML output
      System.out.println(" The XML output is:\n"+str);
      // Always close the query to get rid of any resources..
     qry.close();
     }catch(SQLException e){
      System.out.println(e.toString());
     }
   }

   // Get the connection given the user name and 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;
   }
}
```

How to Run This Program

To run this program:

1. Store the code in a file called `testXMLSQL.java`
2. Compile it using `javac`, the Java compiler
3. Execute it by specifying: `java testXMLSQL`

You must have the `CLASSPATH` pointing to this directory for the Java executable to find the class. Alternatively use various visual Java tools including Oracle JDeveloper to compile and run this program. When run, this program prints out the XML file to the screen.

XSU Generating XML Example 2: Generating DOM from Table employees (Java)

DOM represents an XML document in a parsed tree-like form. Each XML entity becomes a DOM node. Thus XML elements and attributes become DOM nodes while their children become child nodes. To generate a DOM tree from the XML generated by XSU, you can directly request a DOM document from XSU, as it saves the overhead of having to create a string representation of the XML document and then parse it to generate the DOM tree.

XSU calls the parser to directly construct the DOM tree from the data values. The following example illustrates how to generate a DOM tree. The example steps through the DOM tree and prints all the nodes one by one.

```
import org.w3c.dom.*;
import oracle.xml.parser.v2.*;
import java.sql.*;
import oracle.xml.sql.query.OracleXMLQuery;
import java.io.*;

 class domTest{

   public static void main(String[] argv)
   {
      try{
      // create the connection
      Connection conn  = getConnection("hr","hr");

      // Create the query class.
   OracleXMLQuery qry = new OracleXMLQuery(conn, "select * from employees");

      // Get the XML DOM object. The actual type is the Oracle Parser's DOM
      // representation. (XMLDocument)
      XMLDocument domDoc = (XMLDocument)qry.getXMLDOM();

      // Print the XML output directly from the DOM
      domDoc.print(System.out);

      // If you instead want to print it to a string buffer you can do this.
      StringWriter s = new StringWriter(10000);
      domDoc.print(new PrintWriter(s));
      System.out.println(" The string version ---> "+s.toString());

      qry.close();   // Allways close the query!!
      }catch(Exception e){
        System.out.println(e.toString());
      }
    }
    // Get the connection given the user name and password..!
    private static Connection getConnection(String user, String passwd)
      throws SQLException
    {
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
      Connection conn =
          DriverManager.getConnection("jdbc:oracle:oci:@",user,passwd);
     return conn;
   }
}
```

Paginating Results: skipRows and maxRows

In the examples shown so far, XML SQL Utility (XSU) takes the `ResultSet` or the query, and generates the whole document from all the rows of the query. To obtain 100 rows at a time, you then have to fire off different queries to get the first 100 rows, the next 100, and so on. Also it is not possible to skip the first five rows of the query and then generate the result.

To obtain those results, use the XSU `skipRows` and `maxRows` parameter settings:

- `skipRows` parameter, when set, forces the generation to skip the desired number of rows before starting to generate the result.
- `maxRows` limits the number of rows converted to XML.

For example, if you set `skipRows` to a value of 5 and `maxRows` to a value of 10, then XSU skips the first 5 rows, then generates XML for the next 10 rows.

Keeping the Object Open for the Duration of the User's Session

In Web scenarios, you may want to keep the query object open for the duration of the user's session. For example, consider the case of a Web search engine that gives the results of a user's search in a paginated fashion. The first page lists 10 results, the next page lists 10 more results, and so on.

To achieve this, request XSU to convert 10 rows at a time and keep the ResultSet state active, so that the next time you ask XSU for more results, it starts generating from the place the last generation finished.

See Also: "XSU Generating XML Example 3: Paginating Results: (Java)" on page 7-21

When the Number of Rows or Columns in a Row Is Too Large

There is also the case when the number of rows, or number of columns in a row are very large. In this case, you can generate multiple documents each of a smaller size. These cases can be handled by using the `maxRows` parameter and the `keepObjectOpen` function.

keepObjectOpen Function

Typically, as soon as all results are generated, `OracleXMLQuery` internally closes the `ResultSet`, if it created one using the SQL query string given, since it assumes you no longer want any more results. However, in the case described earlier, to maintain that state, you need to call the `keepObjectOpen` function to keep the cursor active. See the following example.

XSU Generating XML Example 3: Paginating Results: (Java)

This example shows how you can use the XSU for Java API to generate an XML page:

```
import oracle.sql.*;
import oracle.jdbc.*;

import oracle.xml.sql.*;
import oracle.xml.sql.query.*;
import oracle.xml.sql.dataset.*;
import oracle.xml.sql.docgen.*;

import java.sql.*;
import java.io.*;

public class b
{
  public static void main(String[] args) throws Exception
  {

    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    Connection conn =
      DriverManager.getConnection"jdbc:oracle:oci:@", "hr", "hr"();

    Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                          ResultSet.CONCUR_READ_ONLY);

    String sCmd = "SELECT FIRST_NAME, LAST_NAME FROM HR.EMPLOYEES";
    ResultSet rs = stmt.executeQuery(sCmd);

    OracleXMLQuery xmlQry = new OracleXMLQuery(conn, rs);
    xmlQry.keepObjectOpen(true);
    //xmlQry.setRowIdAttrName("");
    xmlQry.setRowsetTag("ROWSET");
    xmlQry.setRowTag("ROW");
    xmlQry.setMaxRows(20);

    //rs.beforeFirst();
    String sXML = xmlQry.getXMLString();
    System.out.println(sXML);
  }
}
```

Generating XML from ResultSet Objects

You saw how you can supply a SQL query and get the results as XML. In the last example, you retrieved paginated results. However in Web cases, you may want to retrieve the previous page and not just the next page of results. To provide this scrollable functionality, you can use the Scrollable `ResultSet`. Use the `ResultSet` object to move back and forth within the result set and use XSU to generate the XML each time. The following example illustrates how to do this.

XSU Generating XML Example 4: Generating XML from JDBC ResultSets (Java)

This example shows you how to use the JDBC `ResultSet` to generate XML. Note that using the `ResultSet` might be necessary in cases that are not handled directly by

XSU, for example, when setting the batch size, binding values, and so on. This example extends the previously defined `pageTest` class to handle any page.

```
public class pageTest
{
   Connection conn;
   OracleXMLQuery qry;
    Statement stmt;
   ResultSet rset;
   int lastRow = 0;

   public pageTest(String sqlQuery)
   {
       try{
           conn  = getConnection("hr","hr");
           stmt = conn.createStatement();// create a scrollable Rset
           ResultSet rset = stmt.executeQuery(sqlQuery); // get the result set.
           qry = new OracleXMLQuery(conn,rset); // create an OracleXMLQuery
                                               // instance
           qry.keepCursorState(true); // Don't lose state after the first fetch
           qry.setRaiseNoRowsException(true);
           qry.setRaiseException(true);
       }
     catch (Exception e )
     {
         e.printStackTrace(System.out);
     }
   }

    // Get the connection given the user name and password..!
    private static Connection getConnection(String user, String passwd)
      throws SQLException
    {
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
      Connection conn =
          DriverManager.getConnection("jdbc:oracle:oci:@",user,passwd);
     return conn;
   }

   // Returns the next XML page..!
   public String getResult(int startRow, int endRow)
   {
     qry.setMaxRows(endRow-startRow); // set the max # of rows to retrieve..!
     return qry.getXMLString();
   }

   // Function to still perform the next page.
   public String nextPage()
   {
     String result = getResult(lastRow,lastRow+10);
     lastRow+= 10;
     return result;
   }

   public void close() throws SQLException
   {
     stmt.close();   // close the statement..
     conn.close();   // close the connection
     qry.close();    // close the query..
   }
   public static void main(String[] argv)
   {
     String str;
     try{
         pageTest test = new pageTest("select * from employees");

         int i = 0;
         // Get the data one page at a time..!!!!!
         while ((str = test.getResult(i,i+10))!= null)
             {
                 System.out.println(str);
                 i+= 10;
             }
         test.close();
     }
     catch (Exception e )
     {
         e.printStackTrace(System.out);
     }
   }
}
```

XSU Generating XML Example 5: Generating XML from Procedure Return Values

The `OracleXMLQuery` class provides XML conversion only for query strings or `ResultSets`. But in your application if you have PL/SQL procedures that return REF cursors, how do you do the conversion?

In this case, you can use the earlier-mentioned `ResultSet` conversion mechanism to perform the task. REF cursors are references to cursor objects in PL/SQL. These cursor objects are valid SQL statements that can be iterated upon to get a set of values. These REF cursors are converted into `OracleResultSet` objects in the Java world.

You can execute these procedures, get the `OracleResultSet` object, and then send that to the `OracleXMLQuery` object to get the desired XML.

Consider the following PL/SQL function that defines a REF cursor and returns it:

```
CREATE OR REPLACE PACKAGE BODY testRef IS

  function testRefCur RETURN empREF is
  a empREF;
  begin
      OPEN a FOR select * from hr.employees;
      return a;
  end;
end;
/
```

Every time this function is called, it opens a cursor object for the query, `select * from employees` and returns that cursor instance. To convert this to XML, you do the following:

```
import org.w3c.dom.*;
import oracle.xml.parser.v2.*;
import java.sql.*;
import oracle.jdbc.*;
import oracle.xml.sql.query.OracleXMLQuery;
import java.io.*;
public class REFCURtest
{
   public static void main(String[] argv)
     throws SQLException
   {
      String str;
      Connection conn  = getConnection("hr","hr"); // create connection

      // Create a ResultSet object by calling the PL/SQL function
      CallableStatement stmt =
         conn.prepareCall("begin ? := testRef.testRefCur(); end;");

      stmt.registerOutParameter(1,OracleTypes.CURSOR); // set the define type

      stmt.execute();   // Execute the statement.
      ResultSet rset = (ResultSet)stmt.getObject(1);  // Get the ResultSet

      OracleXMLQuery qry = new OracleXMLQuery(conn,rset); // prepare Query class
      qry.setRaiseNoRowsException(true);
      qry.setRaiseException(true);
      qry.keepCursorState(true);        // set options (keep the cursor active.
      while ((str = qry.getXMLString())!= null)
           System.out.println(str);

      qry.close();    // close the query..!

      // Note since we supplied the statement and resultset, closing the
      // OracleXMLquery instance will not close these. We need to
      // explicitly close this ourselves..!
      stmt.close();
      conn.close();
   }
    // Get the connection given the user name and password..!
    private static Connection getConnection(String user, String passwd)
      throws SQLException
    {
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
      Connection conn =
          DriverManager.getConnection("jdbc:oracle:oci:@",user,passwd);
     return conn;
   }

}
```

To apply the stylesheet, on the other hand, use the `applyStylesheet()` command. This forces the stylesheet to be applied before generating the output.

Raising NoRowsException

When there are no rows to process, XSU simply returns a null string. However, it might be desirable to get an exception every time there are no more rows present, so that the application can process this through exception handlers. When the `setRaiseNoRowsException`() is set, then whenever there are no rows to generate for the output XSU raises an `oracle.xml.sql.OracleXMLSQLNoRowsException`. This is a runtime exception and need not be caught unless needed.

XSU Generating XML Example 6: No Rows Exception (Java)

The following code extends the previous examples to use the exception instead of checking for null strings:

```
public class pageTest {
    .... // rest of the class definitions....

   public static void main(String[] argv)
   {
     pageTest test = new pageTest("select * from employees");

     test.qry.setRaiseNoRowsException(true); // ask it to generate exceptions
     try
     {
        while(true)
         System.out.println(test.nextPage());
     }
     catch(oracle.xml.sql.OracleXMLSQLNoRowsException e)
     {
       System.out.println(" END OF OUTPUT ");
       try{
           test.close();
       }
       catch ( Exception ae )
       {
           ae.printStackTrace(System.out);
       }
     }
   }
}
```

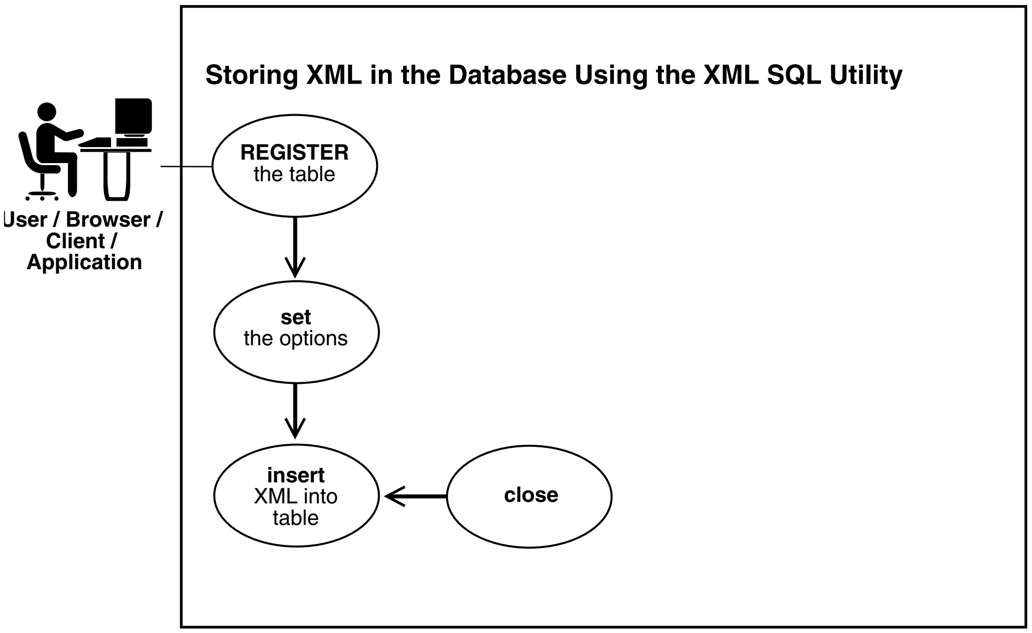
Note: Notice how the condition to check the termination changed from checking if the result is `NULL` to an exception handler.

Storing XML Back in the Database Using XSU OracleXMLSave

Now that you have seen how queries can be converted to XML, here is how you can put the XML back into the tables or views using XSU. The class `oracle.xml.sql.dml.OracleXMLSave` provides this functionality. It has methods to insert XML into tables, update existing tables with the XML document, and delete rows from the table based on XML element values.

In all these cases the given XML document is parsed, and the elements are examined to match tag names to column names in the target table or view. The elements are converted to the SQL types and then bound to the appropriate statement. The process for storing XML using XSU is shown in Figure 7–6.

Figure 7–6 Storing XML in the Database Using XSU


Storing XML in the Database Using the XML SQL Utility
User / Browser / Client / Application
REGISTER the table
set the options
insert XML into table
close

Consider an XML document that contains a list of ROW elements, each of which constitutes a separate DML operation, namely, `INSERT`, `UPDATE`, or `DELETE` on the table or view.

Insert Processing Using XSU (Java API)

To insert a document into a table or view, simply supply the table or the 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 the columns of the table or view and an absent element is treated as a `NULL` value. The following example shows you how the XML document generated from the `employees` table, can be stored in the table with relative ease.

XSU Inserting XML Example 7: Inserting XML Values into All Columns (Java)

This example inserts XML values into all columns:

```
// This program takes as an argument the file name, or a url to
// a properly formated XML document and inserts it into the HR.EMPLOYEES table.
import java.sql.*;
import oracle.xml.sql.dml.OracleXMLSave;
public class testInsert
{
   public static void main(String argv[])
     throws SQLException
  {
     DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
     Connection conn =
          DriverManager.getConnection("jdbc:oracle:oci:@","hr","hr");

     OracleXMLSave sav = new OracleXMLSave(conn, "employees");
     sav.insertXML(sav.getURL(argv[0]));
     sav.close();
   }
}
```

An `INSERT` statement of the form:

```
INSERT INTO hr.employees (employee_id, last_name, job_id, manager_id,
            hire_date, salary, department_id) VALUES(?,?,?,?,?,?,?);
```

is generated, and the element tags in the input XML document matching the column names are matched and their values bound.

If you store the following XML document:

```
<?xml version='1.0'?>
<ROWSET>
 <ROW num="1">
    <EMPLOYEE_ID>7369</EMPLOYEE_ID>
    <LAST_NAME>Smith</LAST_NAME>
    <JOB_ID>CLERK</JOB_ID>
    <MANAGER_ID>7902</MANAGER_ID>
    <HIRE_DATE>12/17/1980 0:0:0</HIRE_DATE>
    <SALARY>800</SALARY>
    <DEPARTMENT_ID>20</DEPARTMENT_ID>
 </ROW>
  <!-- additional rows ... -->
</ROWSET>
```

to a file and specify the file to the program described earlier, you get a new row in the `employees` table containing the values `7369, Smith, CLERK, 7902, 12/17/1980,800,20` for the values named. Any element absent inside the row element is taken as a NULL value.

XSU Inserting XML Example 8: Inserting XML Values into Columns (Java)

In certain cases, you may not want to insert values into *all* columns. This may be true when the group of values that you are getting is not the complete set and you need triggers or default values to be used for the rest of the columns. The code following shows how this can be done.

Assume that you are getting the values only for the employee number, name, and job and that the salary, manager, department number, and hire date fields are filled in automatically. First create a list of column names that you want the `INSERT` statement to work on and then pass it to the `OracleXMLSave` instance.

```
import java.sql.*;
import oracle.xml.sql.dml.OracleXMLSave;
public class testInsert
{
   public static void main(String argv[])
     throws SQLException
   {
      Connection conn = getConnection("hr","hr");
      OracleXMLSave sav = new OracleXMLSave(conn, "hr.employees");

      String [] colNames = new String[3];
      colNames[0] = "EMPLOYEE_ID";
      colNames[1] = "LAST_NAME";
      colNames[2] = "JOB_ID";

      sav.setUpdateColumnList(colNames); // set the columns to update..!

      // Assume that the user passes in this document as the first argument!
      sav.insertXML(sav.getURL(argv[0]));
      sav.close();
   }
   // Get the connection given the user name and password..!
    private static Connection getConnection(String user, String passwd)
      throws SQLException
    {
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
      Connection conn =
          DriverManager.getConnection("jdbc:oracle:oci:@",user,passwd);
     return conn;
   }
}
```

An `INSERT` statement is generated

```
INSERT INTO hr.employees (employee_id, last_name, job_id) VALUES (?, ?, ?);
```

In the preceding example, if the inserted document contains values for the other columns (`HIRE_DATE`, and so on), those are ignored. Also an insert operation is performed for each `ROW` element that is present in the input. These inserts are batched by default.

Update Processing Using XSU (Java API)

Now that you know how to insert values into the table from XML documents, see how you can update only *certain* values. In an XML document, to update the salary of an employee and the department that they work in:

```
<ROWSET>
 <ROW num="1">
   <EMPLOYEE_ID>7369</EMPLOYEE_ID>
   <SALARY>1800</SALARY>
   <DEPARTMENT_ID>30</DEPARTMENT_ID>
 </ROW>
 <ROW>
    <EMPLOYEE_ID>2290</EMPLOYEE_ID>
    <SALARY>2000</SALARY>
    <HIRE_DATE>12/31/1992</HIRE_DATE>
  <!-- additional rows ... -->
</ROWSET>
```

You can use the XSU to update the values. For updates, you must supply XSU with the list of key column names. These form part of the `WHERE` clause in the `UPDATE` statement. In the `employees` table shown earlier, employee number (`employee_id`) column forms the key. Use this for updates.

XSU Updating XML Example 9: Updating a Table Using the keyColumns (Java)

This example updates `table`, `emp`, using `keyColumns`:

```
import java.sql.*;
import oracle.xml.sql.dml.OracleXMLSave;
public class testUpdate
{
   public static void main(String argv[])
     throws SQLException
   {
      Connection conn = getConnection("hr","hr");
      OracleXMLSave sav = new OracleXMLSave(conn, "hr.employees");
      String [] keyColNames = new String[1];
      keyColNames[0] = "EMPLOYEE_ID";
      sav.setKeyColumnList(keyColNames);

      // Assume that the user passes in this document as the first argument!
      sav.updateXML(sav.getURL(argv[0]));
      sav.close();
   }
   // Get the connection given the user name and password..!
    private static Connection getConnection(String user, String passwd)
      throws SQLException
    {
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
      Connection conn =
          DriverManager.getConnection("jdbc:oracle:oci:@",user,passwd);
     return conn;
   }
}
```

In this example, two UPDATE statements are generated. For the first ROW element, you generate an UPDATE statement to update the SALARY and HIRE_DATE fields as follows:

```
UPDATE hr.employees SET salary = 2000 AND hire_date = 12/31/1992 WHERE employee_id = 2290;
```

For the second ROW element:

```
UPDATE hr.employees SET salary = 2000 AND hire_date = 12/31/1992 WHERE employee_id = 2290;
```

XSU Updating XML Example 10: Updating a Specified List of Columns (Java)

You may want to specify a *list* of columns to update. This speeds the processing since the same UPDATE statement can be used for all the ROW elements. Also you can 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, it will be treated as NULL.

If you know that all the elements to be updated are the same for all the ROW elements in the XML document, you can use the setUpdateColumnNames() function to set the list of columns to update.

```
import java.sql.*;
import oracle.xml.sql.dml.OracleXMLSave;
public class testUpdate
{
   public static void main(String argv[])
     throws SQLException
   {
      Connection conn = getConnection("hr","hr");
      OracleXMLSave sav = new OracleXMLSave(conn, "hr.employees");

      String [] keyColNames = new String[1];
      keyColNames[0] = "EMPLOYEE_ID";
      sav.setKeyColumnList(keyColNames);

      // you create the list of columns to update..!
      // Note that if you do not supply this, then for each ROW element in the
      // XML document, you would generate a new update statement to update all
      // the tag values (other than the key columns)present in that element.
      String[] updateColNames = new String[2];
      updateColNames[0] = "SALARY";
      updateColNames[1] = "JOB_ID";
      sav.setUpdateColumnList(updateColNames); // set the columns to update..!

      // Assume that the user passes in this document as the first argument!
      sav.updateXML(sav.getURL(argv[0]));
      sav.close();
   }
   // Get the connection given the user name and password..!
    private static Connection getConnection(String user, String passwd)
      throws SQLException
    {
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
      Connection conn =
          DriverManager.getConnection("jdbc:oracle:oci:@",user,passwd);
     return conn;
   }
}
```

Delete Processing Using XSU (Java API)

When deleting from XML documents, you can set the list of key columns. These columns are used in the `WHERE` clause of the `DELETE` statement. If the key column names are not supplied, then a new `DELETE` statement is created for each `ROW` element of the XML document, where the list of columns in the `WHERE` clause of the `DELETE` statement will match those in the ROW element.

XSU Deleting XML Example 11: Deleting Operations Per Row (Java)

Consider this delete example:

```
import java.sql.*;
import oracle.xml.sql.dml.OracleXMLSave;
public class testDelete
{
   public static void main(String argv[])
     throws SQLException
   {
      Connection conn = getConnection("hr","hr");
      OracleXMLSave sav = new OracleXMLSave(conn, "hr.employees");

      // Assume that the user passes in this document as the first argument!
      sav.deleteXML(sav.getURL(argv[0]));
      sav.close();
   }
   // Get the connection given the user name and password..!
    private static Connection getConnection(String user, String passwd)
      throws SQLException
    {
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
      Connection conn =
          DriverManager.getConnection("jdbc:oracle:oci:@",user,passwd);
     return conn;
   }
}
```

Using the same XML document shown previously for the update example, you get two `DELETE` statements:

```
DELETE FROM hr.employees WHERE employee_id=7369 AND salary=1800 AND department_id=30;
DELETE FROM hr.employees WHERE employee_id=2200 AND salary=2000 AND hire_date=12/31/1992;
```

The `DELETE` statements were formed based on the tag names present in each `ROW` element in the XML document.

XSU Deleting XML Example 12: Deleting Specified Key Values (Java)

If instead, you want the `DELETE` statement to only use the key values as predicates, you can use the `setKeyColumn` function to set this.

```
import java.sql.*;
import oracle.xml.sql.dml.OracleXMLSave;
public class testDelete
{
   public static void main(String argv[])
     throws SQLException
   {
      Connection conn = getConnection("hr","hr");
      OracleXMLSave sav = new OracleXMLSave(conn, "hr.employees");

      String [] keyColNames = new String[1];
      keyColNames[0] = "EMPLOYEE_ID";
      sav.setKeyColumnList(keyColNames);

      // Assume that the user passes in this document as the first argument!
      sav.deleteXML(sav.getURL(argv[0]));
      sav.close();
   }
   // Get the connection given the user name and password..!
    private static Connection getConnection(String user, String passwd)
      throws SQLException
    {
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
      Connection conn =
          DriverManager.getConnection("jdbc:oracle:oci:@",user,passwd);
     return conn;
   }
}
```

Here is the single generated `DELETE` statement:

```
DELETE FROM hr.employees WHERE employee_id=?
```

Advanced XSU Usage Techniques

Here is more information about XSU.

XSU Exception Handling in Java

Exception handling is discussed next.

OracleXMLSQLException Class

XSU catches all exceptions that occur during processing and throws an `oracle.xml.sql.OracleXMLSQLException` which is a runtime exception. The calling program thus does not have to catch this exception all the time, if the program can still catch this exception and do the appropriate action. The exception class provides functions to get the error message and also get the parent exception, if any. For example, the program shown later, catches the run time exception and then gets the parent exception.

OracleXMLNoRowsException Class

This exception is generated when the `setRaiseNoRowsException` is set in the `OracleXMLQuery` class during generation. This is a subclass of the `OracleXMLSQLException` class and can be used as an indicator of the end of row processing during generation.

```
import java.sql.*;
import oracle.xml.sql.query.OracleXMLQuery;

public class testException
{
   public static void main(String argv[])
     throws SQLException
   {
      Connection conn = getConnection("hr","hr");

      // wrong query this will generate an exception
      OracleXMLQuery qry = new OracleXMLQuery(conn, "select * from employees
         where sd = 322323");

      qry.setRaiseException(true); // ask it to raise exceptions..!

      try{
        String str = qry.getXMLString();
      }catch(oracle.xml.sql.OracleXMLSQLException e)
      {
        // Get the original exception
        Exception parent = e.getParentException();
        if (parent instanceof java.sql.SQLException)
        {
           // perform some other stuff. Here you simply print it out..
           System.out.println(" Caught SQL Exception:"+parent.getMessage());
        }
        else
          System.out.println(" Exception caught..!"+e.getMessage());
     }
   }
    // Get the connection given the user name and password..!
    private static Connection getConnection(String user, String passwd)
      throws SQLException
    {
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
      Connection conn =
          DriverManager.getConnection("jdbc:oracle:oci:@",user,passwd);
     return conn;
   }
}
```

Hints for Using XML SQL Utility (XSU)

This section lists XSU hints.

Schema Structure to use with XSU to Store XML

If you have the following XML in your `customer.xml` file:

```
<ROWSET>
 <ROW num="1">
  <CUSTOMER>
   <CUSTOMERID>1044</CUSTOMERID>
   <FIRSTNAME>Paul</FIRSTNAME>
   <LASTNAME>Astoria</LASTNAME>
   <HOMEADDRESS>
    <STREET>123 Cherry Lane</STREET>
    <CITY>SF</CITY>
    <STATE>CA</STATE>
    <ZIP>94132</ZIP>
   </HOMEADDRESS>
  </CUSTOMER>
 </ROW>
</ROWSET>
```

what database schema structure can you use to store this XML with XSU?

Since your example is more than one level deep (that is, it has a nested structure), you can use an object-relational schema. The XML preceding will canonically map to such a schema. An appropriate database schema is:

```
CREATE TYPE address_type AS OBJECT
 (
 street VARCHAR2(40),
 city VARCHAR2(20),
 state VARCHAR2(10),
 zip VARCHAR2(10)
 );
 /
CREATE TYPE customer_type AS OBJECT
 (
customerid NUMBER(10),
firstname VARCHAR2(20),
lastname VARCHAR2(20),
homeaddress address_type
 );
/
CREATE TABLE customer_tab ( customer customer_type);
```

In case you wanted to load `customer.xml` by means of XSU into a relational schema, you can still do it by creating objects in views on top of your relational schema.

For example, you can have a relational table that contains all the following information:

```
CREATE TABLE cust_tab
 ( customerid NUMBER(10),
   firstname VARCHAR2(20),
   lastname VARCHAR2(20),
   street VARCHAR2(40),
   city VARCHAR2(20),
   state VARCHAR2(20),
   zip VARCHAR2(20)
 );
```

Then, you create a customer view that contains a customer object on top of it, as in the following example:

```
CREATE VIEW customer_view AS
SELECT customer_type(customerid, firstname, lastname,
address_type(street,city,state,zip)) customer
FROM cust_tab;
```

Finally, you can flatten your XML using XSLT and then insert it directly into your relational schema. However, this is the least recommended option.

Storing XML Data Across Tables

Currently the XML SQL Utility (XSU) can only store data in a single table. It maps a canonical representation of an XML document into any table or view. But there is a way to store XML with XSU across tables. You can do this using XSLT to transform any document into multiple documents and insert them separately. Another way is to define views over multiple tables (using object views if needed) and then do the insertions into the view. If the view is inherently non-updatable (because of complex joins), then you can use `INSTEAD OF` triggers over the views to do the inserts.

Using XSU to Load Data Stored in Attributes

You have to use XSLT to transform your XML document; that is, you must change the attributes into elements. XSU does assume canonical mapping from XML to a database schema. This takes away a bit from the flexibility, forcing you to sometimes resort to XSLT, but at the same time, in the common case, it does not burden you with having to specify a mapping.

XSU is Case-Sensitive

By default, XSU is case sensitive. You have two options: use the correct case or use the ignoreCase feature.

XSU Cannot Generate the Database Schema from a DTD

Due to a number of shortcomings of the DTD, this functionality is not available. The W3C XML Schema recommendation is finalized, but this functionality is not available yet in XSU.

Thin Driver Connect String Example for XSU

An example of an JDBC thin driver connect string is:

```
jdbc:oracle:thin:user/password@hostname:portnumber:DBSID;
```

Furthermore, the database must have an active TCP/IP listener. A valid OCI connect string is:

```
jdbc:oracle:oci:user/password@hostname
```

XSU and COMMIT After INSERT, DELETE, or UPDATE

Does XML SQL Utility commit after it is done inserting, deleting, or updating? What happens if an error occurs?

By default the XSU executes a number of `INSERT`, `DELETE`, or `UPDATE` statements at a time. The number of statements batch together and executed at the same time can be overridden using the `setBatchSize` feature.

Also, by default XSU does no explicit commits. If `AUTOCOMMIT` is on (default for the JDBC connection), then after each batch of statement executions a commit occurs. You can override this by turning `AUTOCOMMIT` off and then specifying after how many statement executions a commit occurs, which can be done using the `setCommitBatch` feature.

If an error occurs, XSU rolls back to either the state the target table was in before the particular call to XSU, or the state right after the last commit made during the current call to XSU.

Mapping Table Columns to XML Attributes Using XSU

From XSU release 2.1.0 you can map a particular column or a group of columns to an XML attribute instead of an XML element. To achieve this, you have to create an alias for the column name, and prepend the at sign (@) before the name of this alias. For example:

```
* Create a file called select.sql with the following content :
   SELECT empno "@EMPNO", ename, job, hiredate
   FROM emp
   ORDER BY empno

 * Call the XML SQL Utility :
   java OracleXML getXML -user "scott/tiger" \
           -conn "jdbc:oracle:thin:@myhost:1521:ORCL" \
           -fileName "select.sql"

 * As a result, the XML document will look like :
     <?xml version = '1.0'?>
     <ROWSET>
        <ROW num="1" EMPNO="7369">
           <ENAME>SMITH</ENAME>
           <JOB>CLERK</JOB>
           <HIREDATE>12/17/1980 0:0:0</HIREDATE>
        </ROW>
        <ROW num="2" EMPNO="7499">
           <ENAME>ALLEN</ENAME>
           <JOB>SALESMAN</JOB>
           <HIREDATE>2/20/1981 0:0:0</HIREDATE>
        </ROW>
     </ROWSET>
```

Note: All attributes must appear *before* any non-attribute.

Since the XML document is created in a streamed manner, the following query:

```
SELECT ename, empno "@EMPNO", ...
```

does not generate the expected result. It is currently not possible to load XML data stored in attributes. You will still need to use an XSLT transformation to change the attributes into elements. XSU assumes canonical mapping from XML to a database schema.

8 XSQL Pages Publishing Framework

This chapter contains these topics:

XSQL Pages Publishing Framework Overview

The Oracle XSQL Pages publishing framework is an extensible platform for easily publishing XML information in any format you desire. It greatly simplifies combining the power of SQL, XML, and XSLT to publish dynamic Web content based on database information.

Using the XSQL publishing framework, anyone familiar with SQL can create and use declarative templates called "XSQL pages" to:

- Assemble dynamic XML "datagrams" based on parameterized SQL queries, and,
- Transform these "data pages" to produce a final result in any desired XML, HTML, or text-based format using an associated XSLT transformation.

Assembling and transforming information for publishing requires no programming. In fact, most of the common things you will want to do can be easily achieved in a declarative way. However, since the XSQL publishing framework is extensible, if one of the built-in features does not fit your needs, you can easily extend the framework using Java to integrate custom information sources or to perform custom server-side processing.

Using the XSQL Pages framework, the *assembly* of information to be published is cleanly separated from presentation. This simple architectural detail has profound productivity benefits. It enables you to:

- Present the same information in multiple ways, including tailoring the presentation appropriately to the kind of client device making the request (browser, cellular phone, PDA, and so on).

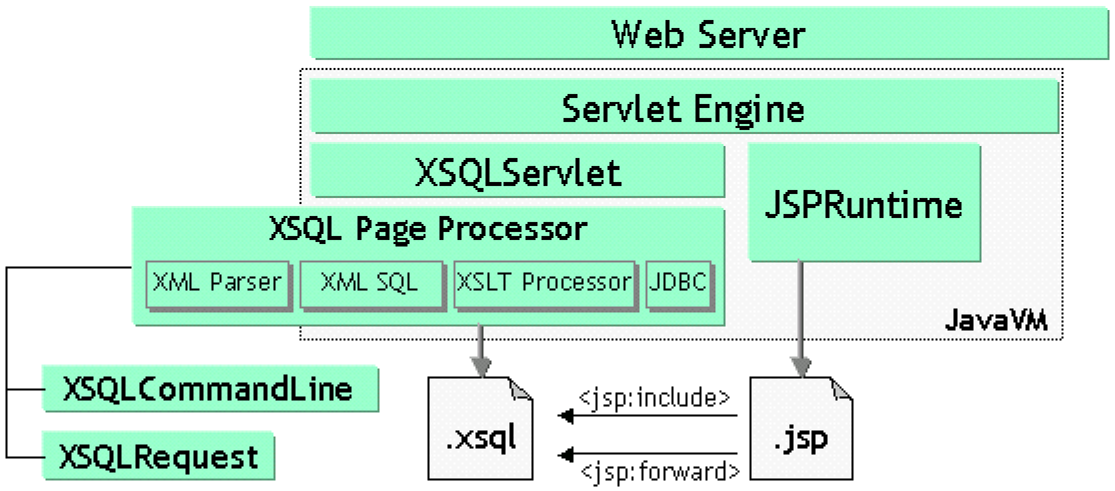
- Reuse information easily by aggregating existing pages into new ones.
- Revise and enhance the presentation independently of the information content being presented.

What Can I Do with Oracle XSQL Pages?

Using server-side templates — known as "XSQL pages" due to their `.xsql` extension — you can publish any information in any format to any device. The XSQL page processor "engine" interprets, caches, and processes the contents of your XSQL page templates. Figure 8–1 illustrates that the core XSQL page processor engine can be "exercised" in four different ways:

- From the command line or in batch using the *XSQL Command-Line Utility*
- Over the Web, using the *XSQL Servlet* installed into your favorite Web server
- As part of JSP applications, using `<jsp:include>` to include a template
- Programmatically, with the `XSQLRequest` object, the engine's Java API

Figure 8–1 Understanding the Architecture of the XSQL Pages Framework



The same XSQL page templates can be used in any or all of these scenarios. Regardless of the means by which a template is processed, the same basic steps occur to produce a result. The XSQL page processor "engine":

1. Receives a request to process an XSQL template
2. Assembles an XML "datagram" using the result of one or more SQL queries
3. Returns this XML "datagram" to the requestor
4. Optionally transforms the "datagram" into any XML, HTML, or text format

During the transformation step in this process, you can use stylesheets that conform to the W3C XSLT 1.0 standard to transform the assembled "datagram" into document formats like:

- 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, like e-mails, SQL scripts, Java programs, and so on
- Arbitrary XML-based document formats

XSQL Pages bring this functionality to you by automating the use of underlying Oracle XML components to solve many common cases without resorting to custom programming. However, when only custom programming will do — as we'll see in the Advanced Topics section of this chapter — you can augment the framework's built-in actions and serializers to assemble the XSQL "datagrams" from any custom source and serialize the datagrams into any desired format, without having to write an entire publishing framework from scratch.

Where Can I Obtain Oracle XSQL Pages?

XSQL Servlet is provided with Oracle and is also available for download from the OTN site.

Where indicated, the examples and demos described in this chapter are also available from OTN.

> **See Also:** XSQL Servlet Release Notes on OTN at `http://www.oracle.com/technology/tech/xml/`

What Is Needed to Run XSQL Pages?

To run the Oracle XSQL Pages publishing framework from the command-line, all you need is a Java VM (1.1.8, 1.2.2, or 1.3). The XSQL Pages framework depends on two underlying components in the Oracle XML Developer's Kit:

- Oracle XML Parser and XSLT Processor (`xmlparserv2.jar`)
- Oracle XML SQL Utility (`xsu12.jar`)

Both of their Java archive files must be present in the CLASSPATH where the XSQL pages framework is running. Since most XSQL pages will connect to a database to query information for publishing, the framework also depends on a JDBC driver. Any JDBC driver is supported, but when connecting to Oracle, it's best to use the Oracle JDBC driver (`classes12.jar`) for maximum functionality and performance.

Lastly, the XSQL publishing engine expects to read its configuration file (by default, named `XSQLConfig.xml`) as a Java resource, so you must include the *directory* where the configuration file resides in the CLASSPATH as well.

To use the XSQL Pages framework for Web publishing, you need a Web server that supports Java Servlets.

> **See Also:** For details on installing the XSQL Servlet on different Web servers, configuring your environment, and running XSQL Servlet, see the XSQL Servlet "Release Notes" on OTN at `http://www.oracle.com/technology/tech/xml`

Security Considerations for XSQL Pages

This section describes best practice security techniques for using the Oracle XSQL Servlet.

Install Your XSQLConfig.xml File in a Safe Directory

The `XSQLConfig.xml` configuration file contains sensitive database username/password information that must be kept secure on the server. This file should not reside in any directory that is mapped to a virtual path of your Web server, nor in any of its subdirectories. The read permissions of the configuration file need only be granted such that the UNIX account that owns the servlet engine can read it.

Failure to follow this recommendation could mean that a user of your site could accidentally, or intentionally, browse the contents of your configuration file.

Disable Default Client Stylesheet Overrides

By default, the XSQL Page Processor allows the user to supply a stylesheet in the request by passing a value for the special `xml-stylesheet` parameter. If you want the stylesheet that is referenced inside your server-side XSQL page to be the only stylesheet that is used, then you can include the `allow-client-style="no"` attribute on the document element of your page. You also can globally change the default setting to disallow client stylesheet overrides by changing a setting in your `XSQLConfig.xml` file. If you do this, then only pages that will allow client stylesheet overrides are ones that include the `allow-client-style="yes"` attribute on their document element.

Be Alert for the Use of Substitution Parameters

With power comes responsibility. Any product such as XSQL Pages 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 important parts of a SQL statement (or at the extreme, the entire 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 the entire SQL statement of an `<xsql:query>` action handler can be supplied as a parameter. This is a powerful capability when in the right users hands, but be aware that if you deploy a similar kind of page to your product system, then the user can execute any query that the database security privileges for the connection associated with the page allows. The demo is setup to use a connection that maps to the SCOTT account, so a user of the "adhoc query demo" can query any data that SCOTT would be allowed to query from the SQL*Plus command line.

Techniques that can be used to make sure your pages are not abused include:

- Making 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.
- Using true bind variables instead of lexical bind variables when substituting single values in a `SELECT` statement. If you need to make syntactic parts of your SQL statement parameterized, then lexical parameters are the only way to proceed. Otherwise, true bind variables are recommended, so that any attempt to pass an invalid value will generate an error instead of producing an unexpected result.

What's New in XSQL Pages Release 10.1

The following list highlights the key new features added in the release 10.1 to the XSQL Pages publishing framework. You can now:

- Easily Work with Multi-Valued Parameters
- Bind Multi-Valued Parameters as Collections in SQL and PL/SQL
- Detect Action Handler Errors and React More Easily to Them
- Conditionally Execute Actions or Include Content

- Use JDBC Datasources from Your Servlet Container
- Provide Custom XSQL Page Request Logging
- Provide Custom XSQL Page Error Handling
- Override the Name of the XSQL Configuration File

The XSQL servlet processor has the following new features in release 10.1:

- Support for Multi-Valued Parameters: This allows users to work with parameters whose values are arrays of strings. The most common scenario where multi-valued parameters occur is when a user submits an HTML form containing multiple occurrences of input controls that share the same name.
- Conditionally Execute Actions or Include Content with `xsql:if-param`: The new `<xsql:if-param>` action enables you to conditionally include the elements and actions that are nested inside it if some condition is true.
- New `Commit="No"` Flag on Actions That Performed an Implicit Commit: The `<xsql:delete-request, xsql:insert-request>`, `xsql:insert-request`, and `<xsql:insert-parameter>` action elements each take a new optional commit attribute to control whether the action does an implicit commit or not.
- Optionally Set an Error Parameter on Any Built-in Action: It is often convenient to know whether an action encountered a non-fatal error during its execution.
- Use Your Servlet Container's DataSource Implementation: As an alternative to defining your named connections in the `XSQLConfig.xml` file, you may now alternatively use the data sources available through your servlet container's implementation of JDBC data sources.
- Provides Custom `XSQLErrorHandler` Implementation: A new interface is introduced in release 1.1. `oracle.xml.xsql.XSQLErrorHandler` allows developers to achieve a programmatic control of how errors are reported to customize the treatment of the errors.
- Provides `Custom XSQLLogger` Implementation: Two new interfaces are introduced in release 10.1: `oracle.xml.xsql.XSQLLoggerFactory` and `oracle.xml.xsql.XSQLLogger` allow developers to log XSQL page requests.
- You can override the Default Name of the `XSQLConfig.xml` file: You can easily provide different configuration files for test and production environments. For example, releases 10.1 introduces two ways to override the file name.
  - By setting the Java System property `xsql.config`
  - By defining a servlet initialization parameter `xsql.config`
- Support for Apache **FOP** 0.20.3: If you need to render PDF output from XSQL pages, this release supports working with the 0.20.3 release candidate of Apache FOP.
- Set Preserve Whitespace Config Option: It is now possible to control whether or not the XSQL Page Processor uses the XML Parser to parse XSQL page templates and XSLT stylesheets with whitespace-preserving mode.

Overview of Basic XSQL Pages Features

In this section, we take a brief look at the most basic features you can exploit in your server-side XSQL page templates:

- Producing XML Datagrams from SQL Queries
- Transforming the XML Datagram into an Alternative XML Format
- Transforming the XML Datagram into HTML for Display

Producing XML Datagrams from SQL Queries

It is extremely easy to serve database information in XML format over the Web using XSQL pages. For example, let us see how simple it is to serve a real-time XML "datagram" from Oracle, of all available flights landing today at JFK airport. Using Oracle JDeveloper, or your favorite text editor, just build an XSQL page template like the one following, and save it in a file named, `AvailableFlightsToday.xsql`:

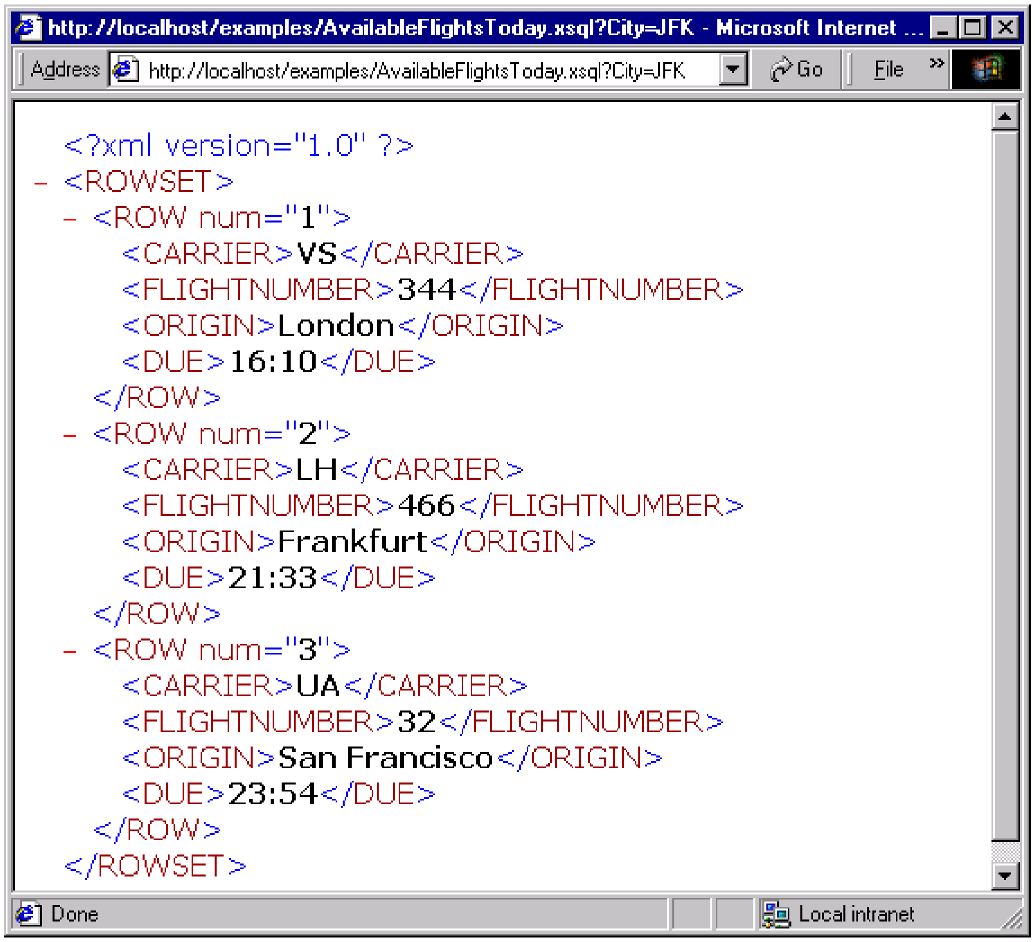
```
<?xml version="1.0"?>
<xsql:query connection="demo" bind-params="City" xmlns:xsql="urn:oracle-xsql">
    SELECT Carrier, FlightNumber, Origin, TO_CHAR(ExpectedTime,'HH24:MI') AS Due
      FROM FlightSchedule
     WHERE TRUNC(ExpectedTime) = TRUNC(SYSDATE) AND Arrived = 'N'
       AND Destination = ?   /* The ? is a bind variable being bound */
  ORDER BY ExpectedTime      /* to the value of the City parameter   */
</xsql:query>
```

With XSQL Servlet properly installed on your Web server, you just need to copy the `AvailableFlightsToday.xsql` file preceding to a directory under your Web server's virtual directory hierarchy. Then you can access the template through a Web browser by requesting the URL:

```
http://yourcompany.com/AvailableFlightsToday.xsql?City=JFK
```

The results of the query in your XSQL page are materialized automatically as XML and returned to the requester. This XML-based "datagram" is typically requested by another server program for processing, but if you are using a browser such as Internet Explorer 5.0, you can directly view the XML result as shown in Figure 8–2.

Figure 8–2 XML Result From XSQL Page (AvailableFlightsToday.xsq) Query


http://localhost/examples/AvailableFlightsToday.xsql?City=JFK - Microsoft Internet ...
Address
http://localhost/examples/AvailableFlightsToday.xsql?City=JFK
Go
File
Done
Local intranet

```
  <?xml version="1.0" ?>
- <ROWSET>
  - <ROW num="1">
      <CARRIER>VS</CARRIER>
      <FLIGHTNUMBER>344</FLIGHTNUMBER>
      <ORIGIN>London</ORIGIN>
      <DUE>16:10</DUE>
    </ROW>
  - <ROW num="2">
      <CARRIER>LH</CARRIER>
      <FLIGHTNUMBER>466</FLIGHTNUMBER>
      <ORIGIN>Frankfurt</ORIGIN>
      <DUE>21:33</DUE>
    </ROW>
  - <ROW num="3">
      <CARRIER>UA</CARRIER>
      <FLIGHTNUMBER>32</FLIGHTNUMBER>
      <ORIGIN>San Francisco</ORIGIN>
      <DUE>23:54</DUE>
    </ROW>
  </ROWSET>
```

Let us take a closer look at the XSQL page template we used. Notice the XSQL page begins with:

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

This is because the XSQL template is itself an XML file (with an `*.xsql` extension) that contains any mix of static XML content and XSQL "action elements". The `AvailableFlightsToday.xsql` example preceding contains no *static* XML elements, and just a single XSQL action element `<xsql:query>`. It represents the simplest useful XSQL page we can build, one that just contains a single query.

Notice that the first (and in this case, only!) element in the page `<xsql:query>` includes a special attribute that declares the `xsql` namespace prefix as a "synonym" for the Oracle XSQL namespace identifier `urn:oracle-xsql`.

```
<xsql:query connection="demo" bind-params="City" xmlns:xsql="urn:oracle-xsql">
```

This first, outermost element — known at the "document element" — also contains a `connection` attribute whose value "demo" is the name of one of the pre-defined connections in the XSQL configuration file (by default, named `XSQLConfig.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 into the configuration file. Setting up these connection definitions is discussed in a later section of this chapter.

Lastly, the `<xsql:query>` element contains a `bind-params` attribute that associates the values of parameters in the request by name to bind parameters represented by question marks in the SQL statement contained inside the `<xsql:query>` tag.

Note that if we wanted to include more than one query on the page, we need to invent an XML element of our own creation to "wrap" the other elements like this:

```
<?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 being bound */
      ORDER BY ExpectedTime  /* to the value of the City parameter   */
  </xsql:query>
  <!-- Other xsql:query actions can go here inside <page> and </page> -->
</page>
```

Notice in this example that the `connection` attribute and the `xsql` namespace declaration *always* go on the document element, while the `bind-params` is specific to the `<xsql:query>` action.

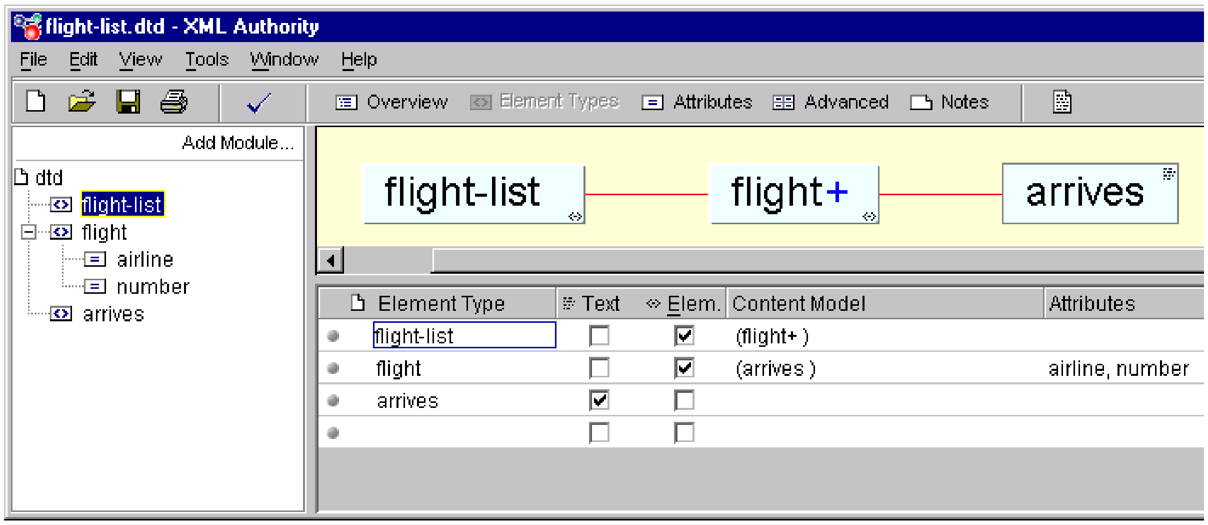
Transforming XML Datagrams into an Alternative XML Format

If the canonical `<ROWSET>` and `<ROW>` XML output from Figure 8–2 is not the XML format you need, then you can associate an XSLT stylesheet to your XSQL page template to transform this XML "datagram" in the server before returning the information in any alternative format desired.

When exchanging data with another program, typically you will agree in advance with the other party on a specific Document Type Definition (DTD) that describes the XML format you will be exchanging. A DTD is in effect, a "schema" definition. It formally defines what XML elements and attributes that a document of that type can have.

Let us assume you are given the `flight-list.dtd` definition and are told to produce your list of arriving flights in a format compliant with that DTD. You can use a visual tool such as Extensibility's "XML Authority" to browse the structure of the flight-list DTD as shown in Figure 8–3.

Figure 8–3 Exploring the "industry standard" flight-list.dtd using Extensibility's XML Authority


flight-list.dtd - XML Authority
File Edit View Tools Window Help
Overview Element Types Attributes Advanced Notes
Add Module...
dtd
flight-list
flight
airline
number
arrives
flight-list
flight+
arrives

Element Type	Text	Elem.	Content Model	Attributes
flight-list	☐	☑	(flight+)	
flight	☐	☑	(arrives)	airline, number
arrives	☑	☐		
	☐	☐		

This shows that the standard XML formats for Flight Lists are:

- `<flight-list>` element, containing one or more…
- `<flight>` elements, having attributes airline and number, each of which contains an…
- `<arrives>` element.

By associating the following XSLT stylesheet, `flight-list.xsl`, with the XSQL page, you can change the default `<ROWSET>` and `<ROW>` format of your arriving flights into the "industry standard" DTD format.

```
<!-- 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 stylesheet is a template that includes the literal elements that you want produced in the resulting document, such as, `<flight-list>`, `<flight>`, and `<arrives>`, interspersed with special XSLT "actions" that allow you to do the following:

- Loop over matching elements in the source document using `<xsl:for-each>`
- Plug in the values of source document elements where necessary using `<xsl:value-of>`
- Plug in the values of source document elements into attribute values using `{something}`

Note two things have been added to the top-level `<flight-list>` element in the stylesheet:

- `xmlns:xsl="http://www.w3.org/1999/XSL/Transform"`

  This defines the XML Namespace (xmlns) named "xsl" and identifies the uniform resource locator 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 that are defined in the XSLT 1.0 specification. Once the namespace is defined, we can then make use of the `<xsl:XXX>` action elements in our stylesheet to loop and plug values in where necessary.

- `xsl:version="1.0"`

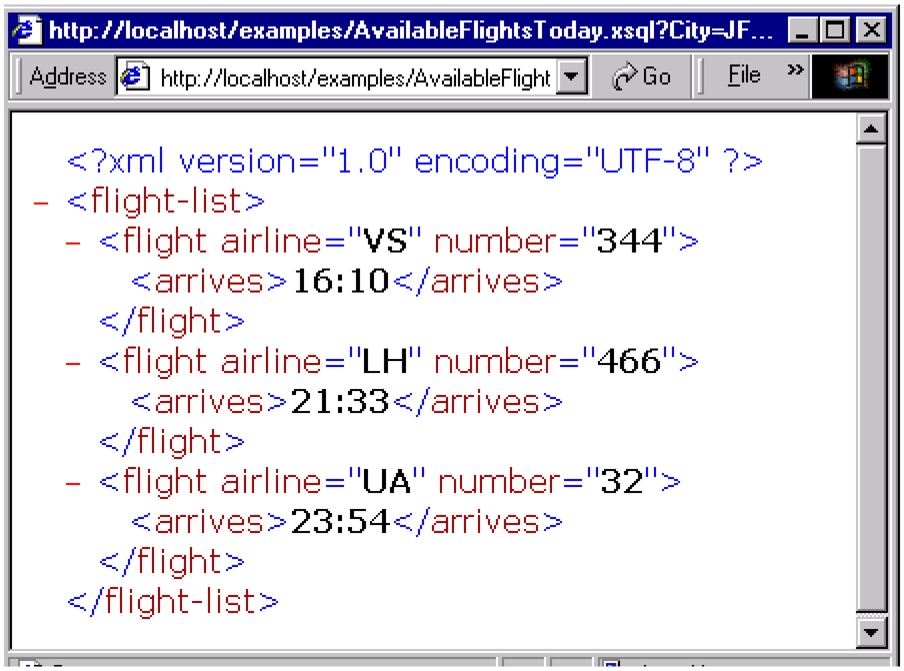
  This attribute identifies the document as an XSLT 1.0 stylesheet. A version attribute is required on all XSLT Stylesheets for them to be valid and recognized by an XSLT Processor.

Associate the stylesheet to your XSQL Page by adding an `<?xml-stylesheet?>` processing instruction to the top of the page as follows:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="flight-list.xsl"?>
<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 ? is a bind variable being bound */
      ORDER BY ExpectedTime  /* to the value of the City parameter   */
</xsql:query>
```

This is the W3C Standard mechanism of associating stylesheets with XML documents (`http://www.w3.org/TR/xml-stylesheet`). Specifying an associated XSLT stylesheet to the XSQL page causes the requesting program or browser to see the XML in the "industry-standard" format as specified by `flight-list.dtd` you were given as shown in Figure 8–4.

Figure 8–4   XSQL Page Results in "industry standard" XML Format


http://localhost/examples/AvailableFlightsToday.xsql?City=JF...
Address
http://localhost/examples/AvailableFlight
Go
File

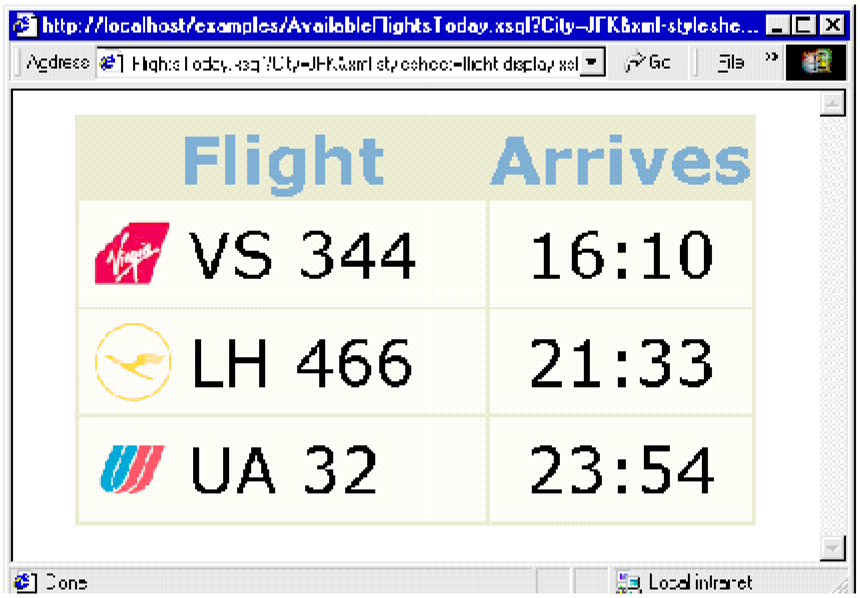
```
<?xml version="1.0" encoding="UTF-8" ?>
- <flight-list>
  - <flight airline="VS" number="344">
      <arrives>16:10</arrives>
    </flight>
  - <flight airline="LH" number="466">
      <arrives>21:33</arrives>
    </flight>
  - <flight airline="UA" number="32">
      <arrives>23:54</arrives>
    </flight>
  </flight-list>
```

Transforming XML Datagrams into HTML for Display

To return the same XML information in HTML instead of an alternative XML format, simply use a different XSLT stylesheet. Rather than producing elements like

<flight-list> and <flight>, your stylesheet produces HTML elements like <table>, <tr>, and <td> instead. The result of the dynamically queried information then looks like the HTML page shown in Figure 8–5. Instead of returning "raw" XML information, the XSQL Page leverages server-side XSLT transformation to format the information as HTML for delivery to the browser.

Figure 8–5 Using an Associated XSLT Stylesheet to Render HTML


http://localhost/examples/AvailableFlightsToday.xsql?City=JFK&xml-styleshe...
Address
FlightsToday.xsql?City=JFK&xml-stylesheet=flight-display.xsl
Go
File
Done
Local intranet

Flight	Arrives
VS 344	16:10
LH 466	21:33
UA 32	23:54

Similar to the syntax of the `flight-list.xsl` stylesheet, the `flight-display.xsl` stylesheet looks like a template HTML page, with `<xsl:for-each>`, `<xsl:value-of>` and attribute value templates like `{DUE}` to plug in the dynamic values from the underlying `<ROWSET>` and `<ROW>` structured XML query results.

```
<!-- 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"/>
                  <xsl:text> </xsl:text>
                  <xsl:value-of select="FLIGHTNUMBER"/>
                </td>
              </tr>
            </table>
          </td>
          <td align="center"><xsl:value-of select="DUE"/></td>
        </tr>
      </xsl:for-each>
    </table></center>
  </body>
</html>
```

Note: The stylesheet looks exactly like HTML, with one tiny difference. It is well-formed HTML. This means that each opening tag is properly closed (for example, `<td>`…`</td>`) and that empty tags use the XML empty element syntax `<br/>` instead of just `<br>`.

You can see that by combining the power of:

- Parameterized SQL statements to select any information you need from our Oracle database,
- Industry-standard XML as a portable, interim data exchange format
- XSLT to transform XML-based "data pages" into any XML- or HTML-based format you need

you can achieve very interesting and useful results quickly. You will see in later sections that what you have seen earlier is just scratching the surface of what you can do using XSQL pages.

See Also: For a detailed introduction to XSLT and a thorough tutorial on how to apply XSLT to many different Oracle database scenarios, see the book *Building Oracle XML Applications*, by Steve Muench, from O'Reilly and Associates.

Setting Up and Using XSQL Pages in Your Environment

You can develop and use XSQL pages in a variety of ways. We start by describing the easiest way to get started, using Oracle JDeveloper, then cover the details you'll need to understand to use XSQL pages in your production environment.

Using XSQL Pages with Oracle JDeveloper

The easiest way to work with XSQL pages during development is to use Oracle JDeveloper. Versions 3.1 and higher of the JDeveloper IDE support color-coded syntax highlighting, XML syntax checking, and easy testing of your XSQL pages. In addition, the JDeveloper 3.2 release supports debugging XSQL pages and adds new wizards to help create XSQL actions.

To create an XSQL page in a JDeveloper project, you can:

- Click the plus icon at the top of the navigator to add a new or existing XSQL page to your project
- Select File | New... and select "XSQL" from the "Web Objects" tab of the gallery

To get assistance adding XSQL action elements like `<xsql:query>` to your XSQL page, place the cursor where you want the new element to go and either:

- Select XSQL Element... from the right mouse menu, or
- Select Wizards | XSQL Element... from the IDE menu.

The XSQL Element wizard takes you through the steps of selecting which XSQL action you want to use, and which attributes you need to provide.

To syntax-check an XSQL page template, you can select *Check XML Syntax...* at any time from the right-mouse menu in the navigator after selecting the name of the XSQL page you'd like to check. If there are any XML syntax errors, they will appear in the message view and your cursor will be brought to the first one.

To test an XSQL page, simply select the page in the navigator and choose *Run* from the right-mouse menu. JDeveloper automatically starts up a local Web-to-go 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. Once you've 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 — and after saving the files in the IDE you can immediately refresh the browser to observe the effect of the changes.

Using JDeveloper, the "XSQL Runtime" library must be added to your project's library list so that the CLASSPATH is properly setup. The IDE adds this entry automatically when you go through the New Object gallery to create a new XSQL page, but you can also add it manually to the project by selecting *Project | Project Properties...* and clicking on the "Libraries" tab.

Setting the CLASSPATH Correctly in Your Production Environment

Outside of the JDeveloper environment, you need to make sure that the XSQL page processor engine is properly configured to run. Oracle comes with the XSQL Servlet pre-installed to the Oracle HTTP Server that accompanies the database, but using XSQL in any other environment, you'll need to ensure that the Java CLASSPATH is setup correctly.

There are three "entry points" to the XSQL page processor:

- `oracle.xml.xsql.XSQLServlet`, the servlet interface
- `oracle.xml.xsql.XSQLCommandLine`, the command-line interface
- `oracle.xml.xsql.XSQLRequest`, the programmatic interface

Since all three of these interfaces, as well as the core XSQL engine itself, are written in Java, they are very portable and very simple to setup. The only setup requirements are to make sure the appropriate JAR files are in the CLASSPATH of the JavaVM that will be running processing the XSQL Pages. The JAR files include:

- `oraclexsql.jar`, the XSQL page processor
- `xmlparserv2.jar`, the Oracle XML Parser for Java v2
- `xsu12.jar`, the Oracle XML SQL utility
- `classes12.jar`, the Oracle JDBC driver

In addition, the *directory* where XSQL Page Processor's configuration file (by default, named `XSQLConfig.xml`) resides must also be listed as a directory in the CLASSPATH.

Putting all this together, if you have installed the XSQL distribution in `C:\xsql`, then your CLASSPATH is:

```
C:\xsql\lib\classes12.classes12.jar;C:\xsql\lib\xmlparserv2.jar;
C:\xsql\lib\xsu12.jar;C:\xsql\lib\oraclexsql.jar;
directory_where_XSQLConfig.xml_resides
```

On UNIX, if you extracted the XSQL distribution into your `/web` directory, the CLASSPATH is:

```
/web/xsql/lib/classes12.jarclasses12.jar:/web/xsql/lib/xmlparserv2.jar:
```

```
/web/xsql/lib/xsu12.jar:/web/xsql/lib/oraclexsql.jar:
directory_where_XSQLConfig.xml_resides
```

To use the XSQL Servlet, one additional setup step is required. You must associate the `.xsql` file extension with the XSQL Servlet Java class `oracle.xml.xsql.XSQLServlet`. How you set the CLASSPATH of the Web server's servlet environment and how you associate a Servlet with a file extension are done differently for each Web server. The XSQL Servlet Release Notes contain detailed setup information for specific Web servers you might want to use with XSQL Pages.

Setting Up the Connection Definitions

XSQL pages refer to database connections by using a short name for the connection defined in the XSQL configuration file. Connection names are defined in the `<connectiondefs>` section of the XSQL configuration file (by default, named `XSQLConfig.xml`) like this:

```
<connectiondefs>
   <connection name="demo">
     <username>scott</username>
     <password>tiger</password>
     <dburl>jdbc:oracle:thin:@localhost:1521:testDB</dburl>
     <driver>oracle.jdbc.driver.OracleDriver</driver>
     <autocommit>false</autocommit>
   </connection>
   <connection name="lite">
     <username>system</username>
     <password>manager</password>
     <dburl>jdbc:Polite:POlite</dburl>
     <driver>oracle.lite.poljdbc.POLJDBCDriver</driver>
    </connection>
</connectiondefs>
```

For each connection, you can specify five pieces of information:

1. `<username>`
2. `<password>`
3. `<dburl>`, the JDBC connection string
4. `<driver>`, the fully-qualified class name of the JDBC driver to use
5. `<autocommit>`, optionally forces the `AUTOCOMMIT` to `TRUE` or `FALSE`

If the `<autocommit>` element is omitted, then the XSQL page processor will use the JDBC driver's default setting of the `AUTOCOMMIT` flag.

Any number of `<connection>` elements can be placed in this file to define the connections you need. 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").

Note: For security reasons, when installing XSQL Servlet on your production Web server, make sure the `XSQLConfig.xml` file does *not* reside in a directory that is part of the Web server's virtual directory hierarchy. Failure to take this precaution risks exposing your configuration information over the Web.

Using the XSQL Command-Line Utility

Often the content of a dynamic page will be based on data that is not frequently changing in your environment. To optimize performance of your Web publishing, you can use operating system facilities to schedule offline processing of your XSQL pages, leaving the processed results to be served statically by your Web server.

You can process any XSQL page from the command line using the XSQL command-line utility. The syntax is:

```
$ java oracle.xml.xsql.XSQLCommandLine xsqlpage [outfile] [param1=value1 ...]
```

If an `outfile` is specified, the result of processing `xsqlpage` is written to it, otherwise the result goes to standard out. Any number of parameters can be passed to the XSQL page processor and are available for reference by the XSQL page being 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. Also can be set 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`

  Used to simulate a particular HTTP User-Agent string from the command line so that an appropriate stylesheet for that User-Agent type will be selected as part of command-line processing of the page.

The `/xdk/java/xsql/bin` directory contains a platform-specific command script to automate invoking the XSQL command-line utility. This script sets up the Java runtime to run `oracle.xml.xsql.XSQLCommandLine` class.

Overview of All XSQL Pages Capabilities

So far we've only seen a single XSQL action element, the `<xsql:query>` action. This is by far the most popular action, but it is not the only one that comes built-in to the XSQL Pages framework. We explore the full set of functionality that you can exploit in your XSQL pages in the following sections.

Using All of the Core Built-in Actions

This section provides a list of the core built-in actions, including a brief description of what each action does, and a listing of all required and optional attributes that each supports.

The <xsql:query> Action

The `<xsql:query>` action element executes a SQL select statement and includes a canonical XML representation of the query's 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.

The syntax for the action is:

```
<xsql:query>
   SELECT Statement
</xsql:query>
```

Any legal SQL select statement is allowed. If the select statement produces no rows, a fallback query can be provided by including a nested <xsql:no-rows-query> element like this:

```
<xsql:query>
  SELECT Statement
  <xsql:no-rows-query>
    SELECT Statement to use if outer query returns no rows
  </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 available on the `<xsql:query>` action element.

By default, the XML produced by a query will reflect the column structure of its resultset, with element names matching the names of the columns. Columns in the result with nested structure like:

- Object Types
- Collection Types
- CURSOR Expressions

produce nested elements that reflect this structure. The result of a typical query containing different types of columns and returning one row might look like this:

```
<ROWSET>
  <ROW id="1">
    <VARCHARCOL>Value</VARCHARCOL>
    <NUMBERCOL>12345</NUMBERCOL>
    <DATECOL>12/10/2001 10:13:22</DATECOL>
    <OBJECTCOL>
       <ATTR1>Value</ATTR1>
       <ATTR2>Value</ATTR2>
    </OBJECTCOL>
    <COLLECTIONCOL>
       <COLLECTIONCOL_ITEM>
         <ATTR1>Value</ATTR1>
         <ATTR2>Value</ATTR2>
       </COLLECTIONCOL_ITEM>
       <COLLECTIONCOL_ITEM>
         <ATTR1>Value</ATTR1>
         <ATTR2>Value</ATTR2>
       </COLLECTIONCOL_ITEM>
    </COLLECTIONCOL>
    <CURSORCOL>
      <CURSORCOL_ROW>
        <COL1>Value1</COL1>
        <COL2>Value2</COL2>
      </CURSORCOR_ROW>
    </CURSORCOL>
  </ROW>
</ROWSET>
```

A `<ROW>` element will repeat for each row in the result set. Your query can use standard SQL column aliasing to rename the columns in the result, and in doing so effectively rename the XML elements that are produced as well. Such column aliasing is *required* for columns whose names otherwise are a illegal names for an XML element.

For example, an `<xsql:query>` action like this:

```
<xsql:query>SELECT TO_CHAR(hire_date,'DD-MON') FROM employees</xsql:query>
```

produces an error because the default column name for the calculated expression will be an illegal XML element name. You can fix the problem with column aliasing like this:

```
<xsql:query>
  SELECT TO_CHAR(hire_date,'DD-MON') as hiredate FROM employees
</xsql:query>
```

The optional attributes listed in Table 8–1 can be supplied to control various aspects of the data retrieved and the XML produced by the `<xsql:query>` action.

Table 8–1 Attributes for <xsql:query>

Attribute Name	Description
`bind-params = "`*`string`*`"`	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
`date-format = "`*`string`*`"`	Date format mask to use for formatted date column/attribute values in XML being queried. Valid values are those documented for the `java.text.SimpleDateFormat` class.
`error-param = "`*`string`*`"`	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.
`error-statement = "`*`boolean`*`"`	If set to `no`, suppresses the inclusion of the offending SQL statement in any `<xsql-error>` element generated. Valid values are `yes` and `no`. The default value is `yes`.
`fetch-size = "`*`integer`*`"`	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`
`id-attribute = "`*`string`*`"`	XML attribute name to use instead of the default `num` attribute for uniquely identifying each row in the result set. If the value of this attribute is the empty string, the row id attribute is suppressed.
`id-attribute-column = "`*`string`*`"`	Case-sensitive name of the column in the result set whose value must be used in each row as the value of the row id attribute. The default is to use the row count as the value of the row id attribute.
`include-schema = "`*`boolean`*`"`	If set to `yes`, includes an inline XML schema that describes the structure of the result set. Valid values are `yes` and `no`. The default value is `no`.
`max-rows = "`*`integer`*`"`	Maximum number of rows to fetch, after optionally skipping the number of rows indicated by the `skip-rows` attribute. If not specified, default is to fetch all rows.

Table 8–1 (Cont.) Attributes for <xsql:query>

Attribute Name	Description
`null-indicator = "boolean"`	Indicates whether to signal that a column's value is NULL by including the `NULL="Y"` attribute on the element for the column. By default, columns with NULL values are omitted from the output. Valid values are `yes` and `no`. The default value is `no`.
`row-element = "string"`	XML element name to use instead of the default `<ROW>` element name for the entire `rowset` of query results. Set to the empty string to suppress generating a containing `<ROW>` element for each row in the result set.
`rowset-element = "string"`	XML element name to use instead of the default `<ROWSET>` element name for the entire `rowset` of query results. Set to the empty string to suppress generating a containing `<ROWSET>` element.
`skip-rows = "integer"`	Number of rows to skip before fetching rows from the result set. Can be combined with `max-rows` for stateless paging through query results.
`tag-case = "string"`	Valid values are `lower` and `upper`. If not specified, the default is to use the case of column names as specified in the query as corresponding XML element names.

The <xsql:dml> Action

You can use the <xsql:dml> action to perform any DML or DDL operation, as well as any PL/SQL block. 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 syntax for the action is:

```
<xsql:dml>
  DML Statement or DDL Statement or PL/SQL Block
</xsql:dml>
```

Table 8–2 lists the optional attributes that you can use on the `<xsql:dml>` action.

Table 8–2 Attributes for <xsql:dml>

Attribute Name	Description
`commit = "boolean"`	If set to `yes`, calls commit on the current connection after a successful execution of the DML statement. Valid values are `yes` and `no`. The default value is `no`.
`bind-params = "string"`	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
`error-param = "string"`	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.
`error-statement = "boolean"`	If set to `no`, suppresses the inclusion of the offending SQL statement in any `<xsql-error>` element generated. Valid values are `yes` and `no`. The default value is `yes`.

The <xsql:ref-cursor-function> Action

The <xsql:ref-cursor-function> action enables you to include the XML results produced by a query whose result set is determined by executing a PL/SQL stored function. 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.

By exploiting PL/SQL's dynamic SQL capabilities, the query can be dynamically and conditionally (or conditionally) constructed by the function before a cursor handle to its result set is returned to the XSQL page processor. As its name implies, the return value of the function being invoked must be of type REF CURSOR.

The syntax of the action is:

```
<xsql:ref-cursor-function>
  [SCHEMA.][PACKAGE.]FUNCTION_NAME(args);
</xsql:ref-cursor-function>
```

With the exception of the fetch-size attribute, the optional attributes available for the <xsql:ref-cursor-function> action are exactly the same as for the <xsql:query> action that are listed Table 8–1.

For example, consider the PL/SQL package:

```
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
   -- Conditionally return a dynamic query as a REF CURSOR
   IF id = 1 THEN
     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;
```

An `<xsql:ref-cursor-function>` can include the dynamic results of the REF CURSOR returned by this function by doing:

```
<xsql:ref-cursor-function>
  DynCursor.DynamicQuery(1);
</xsql:ref-cursor-function>
```

The <xsql:include-owa> Action

The <xsql:include-owa> action enables you to include XML content that has been 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, then the XSQL page 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.

The syntax for the action is:

```
<xsql:include-owa>
   PL/SQL Block invoking a procedure that uses the HTP and HTF (or HTF) packages
</xsql:include-owa>
```

Table 8–3 lists the optional attributes supported by this action.

Table 8–3 Attributes for <xsql:include-owa>

Attribute Name	Description
`bind-params = "`*`string`*`"`	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
`error-param = "`*`string`*`"`	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.
`error-statement = "`*`boolean`*`"`	If set to `no`, suppresses the inclusion of the offending SQL statement in any `<xsql-error>` element generated. Valid values are `yes` and `no`. The default value is `yes`.

Using Bind Variables

To parameterize the results of any of the preceding actions, you can use SQL bind variables. This enables your XSQL page template to produce different results based on the value of parameters passed in the request. To use a bind variable, simply include a question mark anywhere in the statement where bind variables are allowed by SQL. For example, your `<xsql:query>` action might contain the select statement:

```
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
```

Using a question mark to create a bind-variable for the customer id. Whenever the SQL statement is executed in the page, parameter values are bound to the bind variable by specifying the bind-params attribute on the action element. Using the example preceding, we can create an XSQL page that binds the indicated bind variables to the value of the `custid` parameter in the page request like this:

```
<!-- CustomerPortfolio.xsql -->
<portfolio connnection="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 particular customer's portfolio can then be requested by passing the customer id parameter in the request like this:

```
http://yourserver.com/fin/CustomerPortfolio.xsql?custid=1001
```

The value of the `bind-params` attribute is a space-delimited list of parameter names whose left-to-right order indicates the positional bind variable to which its value will be bound in the statement. So, if your SQL statement has five question marks, then your 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 question-mark-indicated bind variable, you simply repeat the name of the parameters in the value of the `bind-params` attribute at the appropriate position. Failure to include exactly as many parameter names in the bind-params attribute as there are question marks in the query, will results in an error when the page is executed.

Bind variables can be used in any action that expects a SQL statement. The following page gives additional examples:

```
<!-- CustomerPortfolio.xsql -->
<portfolio connnection="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>
  <analysis>
    <xsql:include-owa bind-params="custid userCookie">
      BEGIN portfolio_analysis.historical_data(?,5 /* years */, ?); END;
    </xsql:include-owa>
  </analysis>
</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 using a lexical substitution parameter. This enables you to parameterize how the actions behave as well as substitute parts of the SQL statements they perform. Lexical substitution parameters are referenced using the syntax {`@ParameterName`}.

The following example illustrates using two lexical substitution parameters, one which allows the maximum number of rows to be passed in as a parameter, and the other which controls the list of columns to ORDER BY.

```
<!-- 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>
```

This example can then show the XML for a given developer's open bug list by requesting the URL:

```
http://yourserver.com/bug/DevOpenBugs.xsql?dev=smuench&prod=817
```

or using the XSQL Command-Line Utility to request:

```
$ xsql DevOpenBugs.xsql dev=smuench prod=817
```

We close by noting that lexical parameters can also be used to parameterize the XSQL page connection, as well as parameterize the stylesheet that is used to process the page like this:

```
<?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          = ?
       AND status < 80
    ORDER BY {@orderby}
  </xsql:query>
</open-bugs>
```

Providing Default Values for Bind Variables and Parameters

It is often convenient to provide a default value for a bind variable or a substitution parameter directly in the page. This allows the page to be parameterized without requiring the requester to explicitly pass in all the values in each request.

To include a default value for a parameter, simply 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, the XSQL page processor looks for an attribute by the same name on the current action element. If it doesn't 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.

As a simple example, the following page defaults the value of the `max` parameter to 10 for both `<xsql:query>` actions in the page:

```
<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 example defaults the first query to have a `max` of 5, the second query to have a `max` of 7 and the third query to have a `max` of 10.

```
<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>
```

Of course, all of these defaults are overridden if a value of `max` is supplied in the request like:

```
http://yourserver.com/example.xsql?max=3
```

Bind variables respect the same defaulting rules, so a not very useful, yet educational page, like this:

```
<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>
```

returns the XML datagram:

```
<example>
  <rowset>
    <row>
      <somevalue>10</somevalue>
    </row>
  </row>
</example>
```

if the page were requested without any parameters, while a request like:

```
http://yourserver.com/example.xsql?val=3
```

returns:

```
<example>
  <rowset>
    <row>
      <somevalue>3</somevalue>
    </row>
  </row>
</example>
```

To illustrate an important point for bind variables, imagine removing the default value for the `val` parameter from the page by removing the `val` attribute like this:

```
<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>
```

Now a request for the page without supplying any parameters returns:

```
<example>
  <rowset/>
</example>
```

because a bind variable that is bound to a parameter with *neither* a default value *nor* a value supplied in the request will be bound to NULL, causing the WHERE clause in our example page preceding to return no rows.

Understanding the Different Kinds of Parameters

XSQL pages can make use of parameters supplied in the request, as well as page-private parameters whose names and values 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, the value of the `param` parameter is resolved by using:

1. The value of the page-private parameter named `param`, if set, otherwise

2. The value of the request parameter named `param`, if supplied, otherwise
3. The default value provided by an attribute named `param` on the current action element or one of its ancestor elements, otherwise
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, two additional HTTP-specific type of parameters are available to be set and referenced. These are HTTP-Session-level variables and HTTP Cookies. For XSQL pages processed through the XSQL Servlet, the parameter value resolution scheme is augmented as follows. The value of a parameter `param` is resolved by using:

1. The value of the page-private parameter `param`, if set, otherwise
2. The value of the cookie named `param`, if set, otherwise
3. The value of the session variable named `param`, if set, otherwise
4. The value of the request parameter named `param`, if supplied, otherwise
5. The default value provided by an attribute named `param` on the current action element or one of its ancestor elements, otherwise
6. The value NULL for bind variables and the empty string for lexical parameters

The resolution order is arranged this way so that users cannot supply parameter values in a request to override parameters of the same name that have been set in the HTTP session — whose lifetime is the duration of the HTTP session and controlled by your Web server — or set as cookies, which can bet set to "live" across browser sessions.

The <xsql:include-request-params> Action

The `<xsql:include-request-params>` action enables you to include an XML representation of all parameters in the request in your datagram. This is useful if your associated XSLT stylesheet wants to refer to any of the request parameter values by using XPath expressions.

The syntax of the action is:

```
<xsql:include-request-params/>
```

The XML included will have the form:

```
<request>
  <parameters>
    <paramname>value1</paramname>
    <ParamName2>value2</ParamName2>
     ...
  </parameters>
</request>
```

or the form:

```
<request>
  <parameters>
    <paramname>value1</paramname>
    <ParamName2>value2</ParamName2>
     ...
  </parameters>
  <session>
    <sessVarName>value1</sessVarName>
     ...
  </session>
  <cookies>
    <cookieName>value1</cookieName>
     ...
  </cookies>
</request>
```

when processing pages through the XSQL Servlet.

This action has no required or optional attributes.

The <xsql:include-param> Action

The `<xsql:include-param>` action enables you to include an XML representation of a single parameter in your datagram. This is useful if your associated XSLT stylesheet wants to refer to the parameter's value by using an XPath expression.

The syntax of the action is:

```
<xsql:include-param name="paramname" />
```

This `name` attribute is required, and supplies the name of the parameter whose value you want to include. This action has no optional attributes.

If you provide a simple parameter name like this:

```
<xsql:include-param name="productid"/>
```

Then the XML fragment included in the data page will be:

```
<productid>12345</productid>
```

If you use an array-parameter name to indicate that you want to treat the value as an array, like this:

```
<xsql:include-param name="productid[]"/>
```

then the XML fragment will reflect all of the array values like this:

```
<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 will look as if it were a one-element array like this:

```
<productid>
  <value>12345<value>
</productid>
```

The <xsql:include-xml> Action

The `<xsql:include-xml>` action includes the XML contents of a local, remote, or database-driven XML resource into your datagram. The resource is specified either by URL or a SQL statement.

The syntax for this action is:

```
<xsql:include-xml href="URL"/>
```

or

```
<xsql:include-xml>
  SQL select statement selecting a single row containing a single
  CLOB or VARCHAR2 column value
</xsql:include-xml>
```

The URL can be an absolute, http-based URL to retrieve XML from another Web site, or a relative URL. The `href` attribute and the SQL statement are mutually exclusive. If one is provided the other is not allowed.

Table 8–5 lists the attributes supported by this action. Attributes in bold are required.

Table 8–4 Attributes for <xsql:include-xml>

Attribute Name	Description
`bind-params = "string"`	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
`error-param = "string"`	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.

The <xsql:include-posted-xml> Action

The `<xsql:include-posted-xml>` action includes the XML document that has been posted in the request into the XSQL page. If an HTML form is posted instead of an XML document, the XML included will be similar to that included by the `<xsql:include-request-params>` action.

The <xsql:set-page-param> Action

The `<xsql:set-page-param>` action 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.

The syntax for this action is:

```
<xsql:set-page-param name="paramname" value="value"/>
```

or

```
<xsql:set-page-param name="paramname">
  SQL select statement
</xsql:set-page-param>
```

or

```
<xsql:set-page-param name="paramname" xpath="XPathExpression"/>
```

If you use the SQL statement option, a single row is fetched from the result set and the parameter is assigned 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 Page 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.

Once a page-private parameter is set, subsequent action handlers can use this value as a lexical parameter, for example `{@po_id}`, or as a SQL bind parameter value by referencing its name in the `bind-params` attribute of any action handler that supports SQL operations.

If you need to set *several* session parameter values based on the results of a single SQL statement, instead of using the `name` attribute, you can use the `names` attribute and supply a space-or-comma-delimited list of one or more session parameter names. For example:

```
<xsql:set-page-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-page-param>
```

Either the `name` or the `names` attribute is required. The `value` attribute and the contained SQL statement are mutually exclusive. If one is supplied, the other must not be.

Table 8–5 lists the attributes supported by this action. Attributes in **bold** are required.

Table 8–5 Attributes for <xsql:set-page-param>

Attribute Name	Description
name = "string"	Name of the page-private parameter whose value you want to set.
names = "string string ..."	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.
`bind-params = "string"`	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
`error-param = "string"`	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.
`ignore-empty-value = "boolean"`	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`. The default value is `no`.
`treat-list-as-array = "boolean"`	Indicates whether the string-value being 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`.
`iquote-array-values = "boolean"`	If the parameter name being set is a simple-valued parameter name (for example, `myparam`) and if the treat-list-as-array="yes" has been specified, then specifying quote-array-values="yes" will surround each string token with single quotes before separating the values with commas. Valid values are `yes` and `no`. The default value is `no`.

Table 8–5 (Cont.) Attributes for <xsql:set-page-param>

Attribute Name	Description
`xpath = "XPathExpression"`	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 Page Processor.

The <xsql:set-session-param> Action

The `<xsql:set-session-param>` action sets an HTTP session-level parameter to a value. The value of the session-level parameter remains for the lifetime of the current browser user's HTTP session, which is controlled by the Web server. 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.

Since this feature is specific to Java Servlets, this action is only effective if the XSQL page in which it appears is being processed by the XSQL Servlet. If this action is encountered in an XSQL page being processed by the XSQL command-line utility or the `XSQLRequest` programmatic API, this action is a no-op.

The syntax for this action is:

```
<xsql:set-session-param name="paramname" value="value"/>
```

or

```
<xsql:set-session-param name="paramname">
  SQL select statement
</xsql:set-session-param>
```

If you use the SQL statement option, 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 session parameter values based on the results of a single SQL statement, instead of using the name attribute, you can use the names attribute and supply a space-or-comma-delimited list of one or more session parameter names. For example:

```
<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>
```

Either the `name` or the `names` attribute is required. The `value` attribute and the contained SQL statement are mutually exclusive. If one is supplied, the other must not be.

Table 8–6 lists the optional attributes supported by this action.

Table 8–6 Attributes for <xsql:set-session-param>

Attribute Name	Description
`name = "string"`	Name of the session-level variable whose value you want to set.
`names = "string string ..."`	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.

Table 8–6 (Cont.) Attributes for <xsql:set-session-param>

Attribute Name	Description
`bind-params = "string"`	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
`error-param = "string"`	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.
`ignore-empty-value = "boolean"`	Indicates whether the session-level parameter assignment is ignored if the value to which it is being assigned is an empty string. Valid values are `yes` and `no`. The default value is `no`.
`only-if-unset = "boolean"`	Indicates whether the session variable assignment only occurs when the session variable currently does not exists. Valid values are `yes` and `no`. The default value is `no`.

The <xsql:set-cookie> Action

The `<xsql:set-cookie>` action sets an HTTP cookie to a value. By default, the value of the cookie remains for the lifetime of the current browser, but its lifetime can be changed 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 alternatively from the result of a SQL select statement.

Since this feature is specific to the HTTP protocol, this action is only effective if the XSQL page in which it appears is being processed by the XSQL Servlet. If this action is encountered in an XSQL page being processed by the XSQL command-line utility or the `XSQLRequest` programmatic API, this action is a no-op.

The syntax for this action is:

```
<xsql:set-cookie name="paramname" value="value"/>
```

or

```
<xsql:set-cookie name="paramname">
  SQL select statement
</xsql:set-cookie>
```

If you use the SQL statement option, 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, instead of using the name attribute, you can use the names attribute and supply a space-or-comma-delimited list of one or more cookie names. For example:

```
<xsql:set-cookie names="paramname1 paramname2 paramname3">
  SELECT expression_or_column1, expression_or_column2, expression_or_column3
    FROM table
   WHERE clause_identifying_a_single_row
</xsql:set-cookie>
```

Either the `name` or the `names` attribute is required. The value attribute and the contained SQL statement are mutually exclusive. If one is supplied, the other must not be. The number of columns in the select list must match the number of cookies being set or an error message will result.

Table 8–7 lists the optional attributes supported by this action.

Table 8–7 Attributes for <xsql:set-cookie>

Attribute Name	Description
`name = "string"`	Name of the cookie whose value you want to set.
`names = "string string ..."`	Space-or-comma-delimited list of the cookie names whose values you want to set. Either use the `name` or the `names` attribute, but not both.
`bind-params = "string"`	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
`domain = "string"`	Domain in which cookie value is valid and readable. If domain is not set explicitly, then it defaults to the fully-qualified host name (for example, `bigserver.yourcompany.com`) of the document creating the cookie.
`error-param = "string"`	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.
`ignore-empty-value = "boolean"`	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`. The default value is `no`.
`max-age = "integer"`	Sets the maximum age of the cookie in *seconds*. Default is to set the cookie to expire when users current browser session terminates.
`only-if-unset = "boolean"`	Indicates whether the cookie assignment only occurs when the cookie currently does not exists. Valid values are `yes` and `no`. The default value is `no`.
`path = "string"`	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.
`immediate = "boolean"`	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`. The default value is `no`.

The <xsql:set-stylesheet-param> Action

The `<xsql:set-stylesheet-param>` action 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 alternatively 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.

The syntax for this action is:

```
<xsql:set-stylesheet-param name="paramname" value="value"/>
```

or

```
<xsql:set-stylesheet-param name="paramname">
  SQL select statement
```

```
</xsql:set-stylesheet-param>
```

If you use the SQL statement option, 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 stylesheet parameter values based on the results of a single SQL statement, instead of using the `name` attribute, you can use the names attribute and supply a space-or-comma-delimited list of one or more stylesheet parameter names. For example:

```
<xsql:set-stylesheet-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-stylesheet-param>
```

Either the `name` or the `names` attribute is required. The `value` attribute and the contained SQL statement are mutually exclusive. If one is supplied, the other must not be.

Table 8–8 lists the optional attributes supported by this action.

Table 8–8 Attributes for <xsql:set-stylesheet-param>

Attribute Name	Description
`name = "string"`	Name of the top-level stylesheet parameter whose value you want to set.
`names = "string string ..."`	Space-or-comma-delimited list of the top-level stylesheet parameter names whose values you want to set. Either use the `name` or the `names` attribute, but not both.
`bind-params = "string"`	Ordered, space-delimited list of one or more XSQL parameter names whose values will be used to bind to the JDBC bind variable in the appropriate sequential position in the SQL statement.
`error-param = "string"`	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.
`ignore-empty-value = "boolean"`	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`. The default value is `no`.

Working with Array-Valued Parameters

In addition to support for simple-string values, request parameters, session parameters, and page-private parameters may have values that are arrays of strings. To treat to the value of a parameter as an array, you add two empty square brackets to the end of its name. For example, if an HTML form is posted having four occurrences of a input control named `productid`, then to refer to the array-valued productid parameter you use the notation `productid[]`.

If you refer to an array-valued parameter as a lexical substitution parameter, either inside an action handler attribute value or inside the content of an action handler element, its value will be converted to a comma-delimited list of all non-null and non-empty strings in the array in the order that they appear in the array. For example, if you had a page like:

```
<page xmlns:xsql="urn:oracle-xsql">
  <xsql:query>
    select description
      from product
     where productid in ( {@productid[]} )  /* Using lexical parameter */
  </xsql:query>
</page>
```

and the request contains four values for the `productid` parameter, then the `{@productid[]}` lexical substitution expression will be replaced in the query by a string like "`111,222,333,444`".

If you refer to an array-valued parameter without using the array-brackets notation on the end of the name, then the value used will be the value of the ***first*** array entry

Note: Use of a number inside the array brackets is not supported. That is, you can refer to `productid` or `productid[]`, but not `productid[2]`. Only the request parameters, page-private parameters, and session parameters can use string arrays. The `<xsql:set-stylesheet-param>` and `<xsql:set-cookie>` 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 appropriate 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 simply by using the array-brackets notation on the name like this:

```
<!-- Note, param name contains array brackets -->
<xsql:set-page-param name="names[]" value="Tom Jane Joe"/>
```

or similarly for session parameters:

```
<!-- Note, param name contains array brackets -->
<xsql:set-session-param name="dates[]" value="12-APR-1962 15-JUL-1968"/>
```

By default, when the name of the parameter being set is an name with array-brackets, the value will be treated as a space-or-comma-delimited list and tokenized.

The resulting string array value will contain these separate tokens. In the examples earlier, the `names[]` parameter is the string array {"`Tom`", "`Jane`", "`Joe`"} and the `dates[]` parameter is the string array {"`12-APR-1962`", "`15-JUL-1968`"}.

In order to handle strings that contain spaces, the tokenization algorithm first checks the string being tokenized for the presence of any commas. If at least one comma is found in the string, then commas are used as the token delimiter. So, for example, the following action:

```
<!-- Note, param name contains array brackets -->
<xsql:set-page-param name="names[]" value="Tom Jones,Jane York"/>
```

sets the value of the `names[]` parameter to the string array {"`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. So, as in previous releases of XSQL Pages, the following action:

```
<!-- Note, param name does NOT contain array brackets -->
<xsql:set-page-param name="names" value="Tom Jones,Jane York"/>
```

Sets a parameter named `names` to the literal string "`Tom Jones,Jane York`". For convenience, you can optionally *force* the string to be tokenized by including the new `treat-list-as-array="yes"` attribute on the `<xsql:set-page-param>` or `<xsql:set-session-param>` actions. The result will be to assign a comma-delimited string of the tokenized values to the parameter. For example, the action:

```
<!-- Note, param name does NOT contain array brackets -->
<xsql:set-page-param name="names" value="Tom Jane Joe"
                     treat-list-as-array="yes"/>
```

sets the names parameter to the literal string "`Tom,Jane,Joe`".

As a further convenience, when you are setting the value of a simple string-valued parameter and you are tokenizing the value using `treat-list-as-array="yes"`, you can include the `quote-array-values="yes"` attribute to have the comma-delimited values be surrounded by single-quotes. So, an action like this:

```
<!-- Note, param name does NOT contain array brackets -->
<xsql:set-page-param name="names" value="Tom Jones,Jane York,Jimmy"
                     treat-list-as-array="yes"
                     quote-array-values="yes"/>
```

assigns the literal string value "`'Tom Jones','Jane York','Jimmy'`" to the `names` parameter.

Binding Array-Valued Parameters in SQL and PL/SQL Statements

Anywhere in XSQL Pages where string-valued scalar bind variables are supported, you may also bind array-valued parameters by simply using the array-parameter name (for example, `myparam[]`) in the list of parameter names that you supply for the `bind-params` attribute.

This makes it very easy to process array-valued parameters in SQL statements and in PL/SQL procedures. Array-valued parameters are bound as a nested table object type named `XSQL_TABLE_OF_VARCHAR` that you must create in your current schema using the DDL statement:

```
CREATE TYPE xsql_table_of_varchar AS TABLE OF VARCHAR2(2000);
```

While the type *must* have this exact name, `XSQL_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 following PL/SQL stored procedure:

```
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 following page illustrates how to bind two array-valued parameters in a SQL statement that uses this PL/SQL function taking `XSQL_TABLE_OF_VARCHAR`-typed arguments.

```
<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>
```

This produces a resulting XML data page of:

```
<page someNames="aa,bb,cc" someValues="11,22,33">
  <ROWSET>
    <ROW num="1">
      <EXAMPLE>aa=11:bb=22:cc=33</EXAMPLE>
    </ROW>
  </ROWSET>
</page>
```

illustrating that the array-valued `someNames[]` and `someValues[]` parameters were bound as table collection types and the values were iterated over and concatenated together to produce the "`aa=11:bb=22:cc=33`" string value as the function's return value.

You can mix any number of regular parameters and array-valued parameters in your bind-params string. Just use the array-bracket notation for the ones you want to be bound as arrays.

Note: If you try the example earlier and you have not created the `XSQL_TABLE_OF_VARCHAR` type as illustrated earlier, you will receive an error like this:

```
<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>
```

Since the array parameters are bound as nested table collection types, you can use the `TABLE()` operator in combination with the `CAST()` operator in SQL to treat the nested table bind variable value as a table of values to query against. This can be quite a powerful technique to use in sub-select clauses of a SQL statement (but it's not limited to this). The following page illustrates using an array-valued parameter containing employee id's to restrict the rows queried from the familiar EMPLOYEES table in the HR schema.

```
<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>
```

This produces a result like:

```
<page>
  <ROWSET>
    <ROW num="1">
      <NAME>Alana Walsh</NAME>
      <SALARY>3100</SALARY>
    </ROW>
    <ROW num="2">
      <NAME>Kevin Feeny</NAME>
      <SALARY>3000</SALARY>
    </ROW>
  </ROWSET>
</page>
```

These examples have shown using `bind-params` with `<xsql:query>`, but these new features work for `<xsql:dml>`, `<xsql:include-owa>`, `<xsql:ref-cursor-function>`, and any other actions that accept SQL or PL/SQL statements as part of their functionality.

Finally, some users might ask, "Why doesn't XSQL support using PL/SQL index-by tables instead of nested table collection types for binding string-array values?" The simple answer is that PL/SQL index-by-tables do not work with the JDBC Thin driver. They only work using the OCI JDBC driver. By using the nested table collection type `XSQL_TABLE_OF_VARCHAR` we can use the array-valued parameters with both the Thin driver and the OCI driver, without losing any of the programming flexibility of working with the array of values in PL/SQL.

Supplying Multi-Valued Parameters on the Command Line

If you use the `oracle.xml.xsql.XSQLCommandLine` command-line utility to run XSQL pages, you can supply multi-valued parameters to the XSQL page processor by simply including the same parameter name on the command line multiple times like this:

```
java oracle.xml.xsql.XSQLCommandLine SomePage.xsql user=Steve user=Paul user=Mary
```

This will result in having the `user[]` array-valued parameter set as a request parameter to the value {"`Steve`","`Paul`","`Mary`"}.

Supplying Multi-Valued Parameters Programmatically with XSQLRequest

The `XSQLRequest` programmatic API to the XSQL Page engine already takes a `java.util.Dictionary` of named parameters. Typically users have used a `Hashtable` and called its `put(name,value)` method to add `String`-valued parameters to the request. To add multi-valued parameters, simply put a value of type `String[]` instead of type `String`.

Conditionally Executing Actions or Including Content with <xsql:if-param>

The `<xsql:if-param>` action enables you to conditionally include the elements and actions (or actions) that are nested inside it if some condition is true. If the condition evaluates to true, then all nested XML content and actions are included in the page. If the condition evaluates to false, then none of the nested XML content or actions are included (and hence none of the nested actions is executed).

You specify which parameter value will be evaluated by supplying the required `name` attribute. Both simple parameter names as well as array-parameter names are supported.

In addition to the name attribute, you must also pick exactly one of the following five attributes to indicate how the parameter value (or values, in the array case) is tested:

1. `exists="yes"` or `exists="no"`

   If you use `exists="yes"`, then this 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 you use `exists="no"`, then evaluates to true if the parameter does not exist, of 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.

2. `equals="stringValue"`

   This tests whether the named parameter equals the string value provided. By default the comparison is an **exact** string match. For an array-valued parameter, it tests whether any element in the array has the indicated value.

3. `not-equals="stringValue"`

   This tests whether the named parameter does not equal the string value provided. For an array-valued parameter, evaluates to true if none of the elements in the array has the indicated value.

4. `in-list="comma-or-space-separated-list"`

   This tests whether the named parameter matches any of the strings in the provided list. The value of the `in-list` parameter is tokenized into an array using commas as the delimiter if any commas are detected in the string, otherwise using space as the delimiter. For an array-valued parameter, it tests whether any element in the array matches some element in the list.

5. `not-in-list="comma-or-space-separated-list"`

   This tests whether the named parameter does not match any of the strings in the provided list. The value of the `not-in-list` parameter is tokenized into an array using commas as the delimiter if any commas are detected in the string, otherwise using space as the delimiter. For an array-valued parameter, it tests whether none of the elements in the array matches any element in the list.

For the `equals`, `not-equals`, `in-list`, and `not-in-list` tests, by default the comparison is an exact string match. If you want a case-insensitive match, supply the additional `ignore-case="yes"` attribute as well.

As with other XSQL actions, all of the attributes of the `<xsql:if-param>` action can contain lexical substitution parameter expressions (for example, `{@paramName}`) if needed.

Note that any XML content and XSQL action elements (or XSQL action elements) can be nested inside an `<xsql:if-param>`, including other `<xsql:if-param>` elements if needed.

For example, to test whether two different conditions are true, you can use nested `<xsql:if-param>` elements like this:

```
<!--
| Set page param 'foo' to value "bar" if parameter 'a'
| exists, and if parameter 'b' has value equal to "X"
+-->
<xsql:if-param name="a" exists="yes">
  <xsql:if-param name="b" equals="X">
    <xsql:set-page-param name="foo" value="bar"/>
  </xsql:if-param>
</xsql:if-param>
```

Note: If the parameter being tested does not exist, the test evaluates to false.

Optionally Setting an Error Parameter on Any Built-in Action

It is often convenient to know 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 which will get included into your XSQL data page as an `<xsql-error>` element.

Now you can optionally have any built-in XSQL action set a page-private parameter of your choice when that action reports a non-fatal error by using the `error-param` attribute on your action.

For example, to have the parameter named "`dml-error`" set if the statement inside the `<xsql:dml>` action encounters a database error, use an action like this:

```
<xsql:dml error-param="dml-error" bind-params="val">
  insert into yourtable(somecol) values(?)
</xsql:dml>
```

If the execution of this action encounters an error, then the page-private parameter named `dml-error` will be set to the string "`Error`".

If the execution of the action is successful, the error parameter is **not** assigned any value. In the example earlier, this means that if the page-private parameter `dml-error` already exists, it will retain its current value. If it does not exist, it will continue to not exist.

By using this new error parameter in combination with `<xsql:if-param>` you can achieve conditional behavior in your XSQL page template, depending on the success or failure of certain actions. For example, assuming your connection definition sets the `AUTOCOMMIT` flag to `false` on the connection named "demo" in the XSQL configuration file (by default, named `XSQLConfig.xml`), then the following page illustrates how you might rollback the changes made by a previous action if a subsequent action encounters an error.

```
<!-- 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">
    <xsql:dml>rollback</xsql:dml>
  </xsql:if-param>
</page>
```

If you've written any custom action handlers and your custom actions call `reportMissingAttribute()`, `reportError()`, or `reportErrorIncludingStatement()` to report non-fatal action errors, then they will automatically pickup this new feature as well.

Aggregating Information Using <xsql:include-xsql>

The `<xsql:include-xsql>` action makes it very easy to include the results of one XSQL page into another page. This enables you to easily aggregate content from a page that you've already built and find another purpose for it. The examples that follow illustrate two of the most common uses of `<xsql:include-xsql>`.

Assume you have an XSQL page that lists discussion forum categories:

```
<!-- Categories.xsql -->
<xsql:query connection="forum" xmlns:xsql="urn:oracle-xsql">
  SELECT name
    FROM categories
    ORDER BY name
</xsql:query>
```

You can include the results of this page into a page that lists the ten most recent topics in the current forum like this:

```
<!-- TopTenTopics.xsql -->
<top-ten-topics connection="forum" 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 use `<xsql:include-xsql>` to include an existing page to apply an XSLT stylesheet to it as well. So, if we have two different XSLT stylesheets:

- `cats-as-html.xsl`, which renders the topics in HTML, and
- `cats-as-wml.xsl`, which renders the topics in WML

Then one approach for catering to two different types of devices is to create different XSQL pages for each device. We can create:

```
<?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"/>
```

which aggregates `Categories.xsql` and applies the `cats-as-html.xsl` stylesheet, and another page:

```
<?xml version="1.0"?>
<!-- WMLCategories.xsql -->
<?xml-stylesheet type="text/xsl" href="cats-as-html.xsl"?>
<xsql:include-xsql href="Categories.xsql" xmlns:xsql="urn:oracle-xsql"/>
```

which aggregates `Categories.xsql` and applies the `cats-as-wml.xsl` stylesheet for delivering to wireless devices. In this way, we've re-purposed the reusable Categories.xsql page content in two different ways.

If the page being aggregated contains an `<?xml-stylesheet?>` processing instruction, then that stylesheet is applied before the result is aggregated, so using `<xsql:include-xsql>` you can also easily chain the application of XSLT stylesheets together.

When one XSQL page aggregates another page's content using `<xsql:include-xsql>` all of the request-level parameters are visible to the "nested" page. For pages processed by the XSQL Servlet, this also includes session-level parameters and cookies, too. As you expect, none of the aggregating page's *page-private* parameters are visible to the nested page.

Table 8–9 lists the attributes supported by this action. Required attributes are in bold.

Table 8–9 Attributes for <xsql:include-xsql>

Attribute Name	Description
`href = "string"`	Relative or absolute URL of XSQL page to be included.
`error-param = "string"`	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.
`reparse = "boolean"`	Indicates whether output of included XSQL page has to be reparsed before it is included. Useful if included XSQL page is selecting the text of an XML document fragment that the including page wants to treat as elements. Valid values are `yes` and `no`. The default value is `no`.

Including XMLType Query Results

Oracle9*i* introduced the XMLType for use with storing and querying XML-based database content. You can exploit database XML features to produce XML for inclusion in your XSQL pages using one of two techniques:

- `<xsql:query>` handles any query including columns of type XMLType, however it handles XML markup in CLOB/VARCHAR2 columns as literal text.
- `<xsql:include-xml>` *parses* and includes a single CLOB or String-based XML document retrieved from a query

The difference between the two approaches lies in the fact that the `<xsql:include-xml>` action parses the literal XML appearing in a CLOB or

String-value to turn it on the fly into a tree of elements and attributes. On the other hand, using the `<xsql:query>` action, XML markup appearing in CLOB or String valued-columns is left as literal text.

Another difference is that while `<xsql:query>` can handle query results of any number of columns and rows, the `<xsql:include-xml>` is designed to work on a single column of a single row. Accordingly, when using `<xsql:include-xml>`, the SELECT statement that appears 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 XML document will be parsed and included into your XSQL page.

The following example uses nested `xmlagg()` functions to aggregate the results of a dynamically-constructed XML document containing departments and nested employees into a *single* XML "result" document, wrapped in a `<DepartmentList>` element:

```
<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
               )
             )
           )
         ) as result
   from departments d
  order by department_name
</xsql:query>
```

Considering another example, suppose you have a number of <Movie> XML documents stored in a table of `XmlType` called `MOVIES`. Each document might look something like this:

```
<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>
```

```
<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 Oracle XPath query features to extract an aggregate list of all cast members who have received Oscar awards from any movie in the database using a query like this:

```
 select xmlelement("AwardedActors",
           xmlagg(extract(value(m),
                  '/Movie/Cast/*[Award[@From="Oscar"]]')))
  from movies m
```

To include this query result of `XMLType` into your XSQL page, simply paste the query inside an `<xsql:query>` element, and make sure you include an alias for the query expression (for example "as result" following):

```
<xsql:query connection="orcl92" xmlns:xsql="urn:oracle-xsql">
  select xmlelement("AwardedActors",
           xmlagg(extract(value(m),
                  '/Movie/Cast/*[Award[@From="Oscar"]]'))) as result
    from movies m
</xsql:query>

Note that again we use the combination of xmlelement() and xmlagg() to have the
 database aggregate all of the XML fragments identified by the query into
  single, well-formed XML document. The combination of xmlelement() and xmlagg()
 work together to produce a well-formed result like this:
<AwardedActors>
  <Actor>...</Actor>
  <Actress>...</Actress>
</AwardedActors>
```

Notice that you can use the standard XSQL Pages bind variable capabilities in the middle of an XPath expression, too, 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 this:

```
<xsql:query connection="orcl92" xmlns:xsql="urn:oracle-xsql"
            award-from="Oscar" bind-params="award-from">
  /* Using a bind variable in an XPath expression */
  select xmlelement("AwardedActors",
           xmlagg(extract(value(m),
                  '/Movie/Cast/*[Award[@From="'|| ? ||'"]]'))) as result
    from movies m
</xsql:query>
```

Handling Posted Information

In addition to simplifying the assembly and transformation of XML content, the XSQL Pages framework makes it easy to handle posted XML content as well. Built-in actions simplify the handling of posted information from both XML document and HTML forms, and allow that information to be posted directly into a database table using the underlying facilities of the Oracle XML SQL Utility.

The XML SQL Utility provides the ability to data database inserts, updates, and deletes based on the content of an XML document in canonical form with respect to a target table or view. For a given database table, the canonical XML form of its data is given by one row of XML output from a `SELECT * FROM tablename` query against it. Given an XML document in this canonical form, the XML SQL Utility can automate the insert, update, and delete for you. By combining the XML SQL Utility with an XSLT transformation, you can transform XML in any format into the canonical format expected by a given table, and then ask the XML SQL Utility to insert, update, delete the resulting canonical XML for you.

The following built-in XSQL actions make exploiting this capability easy from within your XSQL pages:

- `<xsql:insert-request>`

  Insert the optionally transformed XML document that was posted in the request into a table.Table 8–10 lists the required and optional attributes supported by this action.

- `<xsql:update-request>`

  Update the optionally transformed XML document that was posted in the request into a table or view. Table 8–11 lists the required and optional attributes supported by this action.

- `<xsql:delete-request>`

  Delete the optionally transformed XML document that was posted in the request from a table or view. Table 8–12 lists the required and optional attributes supported by this action.

- `<xsql:insert-param>`

  Insert the optionally transformed XML document that was posted as the value of a request parameter into a table or view. Table 8–13 lists the required and optional attributes supported by this action.

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.

Table 8–10 Attributes for <xsql:insert-request>

Attribute Name	Description
`table = "string"`	Name of the table, view, or synonym to use for inserting the XML information.
`transform = "URL"`	Relative or absolute URL of the XSLT transformation to use to transform the document to be inserted into canonical ROWSET/ROW format.
`columns = "string"`	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.
`commit = "boolean"`	If set to `yes`, calls commit on the current connection after a successful execution of the insert. Valid values are `yes` and `no`. The default value is `yes`.

Table 8–10 (Cont.) Attributes for <xsql:insert-request>

Attribute Name	Description
`commit-batch-size = "integer"`	If a positive, nonzero number N is specified, then after each batch of N inserted records, a commit will be issued. Default batch size is zero (0) if not specified, meaning not to commit interim batches.
`date-format = "string"`	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.
`error-param = "string"`	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.

Table 8–11 Attributes for <xsql:update-request>

Attribute Name	Description
`table = "string"`	Name of the table, view, or synonym to use for inserting the XML information.
`key-columns = "string"`	Space-delimited or comma-delimited list of one or more column names whose values in the posted XML document will be used to identify the existing rows to update.
`transform = "URL"`	Relative or absolute URL of the XSLT transformation to use to transform the document to be inserted into canonical ROWSET/ROW format.
`columns = "string"`	Space-delimited or comma-delimited list of one or more column names whose values will be updated. If supplied, then only these columns will be updated. If not supplied, all columns will be updated, with NULL values for columns whose values do not appear in the XML document.
`commit = "boolean"`	If set to `yes`, calls commit on the current connection after a successful execution of the update. Valid values are `yes` and `no`. The default value is `yes`.
`commit-batch-size = "integer"`	If a positive, nonzero number N is specified, then after each batch of N inserted records, a commit will be issued. Default batch size is zero (0) if not specified, meaning not to commit interim batches.
`date-format = "string"`	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.
`error-param = "string"`	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.

Table 8–12 Attributes for <xsql:delete-request>

Attribute Name	Description
`table = "string"`	Name of the table, view, or synonym to use for inserting the XML information.
`key-columns = "string"`	Space-delimited or comma-delimited list of one or more column names whose values in the posted XML document will be used to identify the existing rows to update.

Table 8–12 (Cont.) Attributes for <xsql:delete-request>

Attribute Name	Description
`transform = "URL"`	Relative or absolute URL of the XSLT transformation to use to transform the document to be inserted into canonical ROWSET/ROW format.
`commit = "boolean"`	If set to `yes`, calls commit on the current connection after a successful execution of the delete. Valid values are `yes` and `no`. The default value is `yes`.
`commit-batch-size = "integer"`	If a positive, nonzero number N is specified, then after each batch of N inserted records, a commit will be issued. Default batch size is zero (0) if not specified, meaning not to commit interim batches.
`error-param = "string"`	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.

Table 8–13 Attributes for <xsql:insert-param>

Attribute Name	Description
`name = "string"`	Name of the parameter whose value contains XML to be inserted.
`table = "string"`	Name of the table, view, or synonym to use for inserting the XML information.
`transform = "URL"`	Relative or absolute URL of the XSLT transformation to use to transform the document to be inserted into canonical ROWSET/ROW format.
`columns = "string"`	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.
`commit = "boolean"`	If set to `yes`, calls commit on the current connection after a successful execution of the insert. Valid values are `yes` and `no`. The default value is `yes`.
`commit-batch-size = "integer"`	If a positive, nonzero number N is specified, then after each batch of N inserted records, a commit will be issued. Default batch size is zero (0) if not specified, meaning not to commit interim batches.
`date-format = "string"`	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.
`error-param = "string"`	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.

Understanding Different XML Posting Options

There are three different ways that the XSQL pages framework can handle posted information.

1. A client program can send an HTTP POST message that targets an XSQL page, whose request body contains an XML document and whose HTTP header reports a `ContentType` of "`text/xml`".

In this case, you can use the `<xsql:insert-request>`, `<xsql:update-request>`, or the `<xsql:delete-request>` action and the content of the posted XML will be insert, updated, or deleted in the target table as indicated. If you transform the posted XML document using an XSLT transformation, the posted XML document is the source document for this transformation.

2. A client program can send 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 and the content of the posted XML parameter value will be inserted in the target table as indicated. If you transform the posted XML document using an XSLT transformation, the XML document in the parameter value is the source document for this transformation.

3. A browser can submit an HTML form with `method="POST"` whose action targets an XSQL page. In this case, by convention the browser sends an HTTP POST message whose request body contains an encoded version of all of the HTML form's fields and their values with a ContentType of "`application/x-www-form-urlencoded`"

   In this case, there request does not contain an XML document, but instead an encoded version of the form parameters. However, to make all three of these cases uniform, the XSQL page processor will (on demand) materialize an XML document from the set of form parameters, session variables, and cookies contained in the request. Your XSLT transformation then transforms this dynamically-materialized XML document into canonical form for insert, update, or delete using `<xsql:insert>`, `<xsql:update-request>`, or `<xsql:delete-request>` respectively.

When working with posted HTML forms, the dynamically materialized XML document will have the following form:

```
<request>
  <parameters>
    <firstparamname>firstparamvalue</firstparamname>
     ...
    <lastparamname>lastparamvalue</lastparamname>
  </parameters>
  <session>
    <firstparamname>firstsessionparamvalue</firstparamname>
      ...
    <lastparamname>lastsessionparamvalue</lastparamname>
  </session>
  <cookies>
    <firstcookie>firstcookievalue</firstcookiename>
       ...
    <lastcookie>firstcookievalue</lastcookiename>
  </cookies>
</request>
```

If multiple parameters are posted with the same name, then they will automatically be "row-ified" to make subsequent processing easier. This means, for example, that a request which posts or includes the following parameters:

- `id` = 101
- `name` = Steve
- `id` = 102
- `name` = Sita

- `operation` = update

Will create a "row-ified" set of parameters like:

```
<request>
  <parameters>
    <row>
      <id>101</id>
      <name>Steve</name>
    </row>
    <row>
      <id>102</id>
      <name>Sita</name>
    </row>
    <operation>update</operation>
  </parameters>
      ...
</request>
```

Since you will need to provide an XSLT stylesheet that transforms this materialized XML document containing the request parameters into canonical format for your target table, it might be useful to build yourself an XSQL page like this:

```
<!--
 | 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, and in the browser you will see the "raw" XML for the materialized XML request document which you can save out and use to develop the XSLT transformation.

Using Custom XSQL Action Handlers

When you need to perform tasks that are not handled by the built-in action handlers, the XSQL Pages framework allows custom actions to be invoked to do virtually any kind of job you need done as part of page processing. Custom actions can supply arbitrary XML content to the data page and perform arbitrary processing. See Writing Custom XSQL Action Handlers later in this chapter for more details on writing custom action handlers in Java. Here we explore how to make use of a custom action handler, once it's already created.

To invoke a custom action handler, use the built-in `<xsql:action>` action element. It has a single, required attribute named `handler` whose value is the fully-qualified Java class name of the action you want to invoke. The class must implement the `oracle.xml.xsql.XSQLActionHandler` interface. For example:

```
<xsql:action handler="yourpackage.YourCustomHandler"/>
```

Any number of additional attribute can be supplied to the handler in the normal way. For example, if the `yourpackage.YourCustomHandler` is expecting a attributes named `param1` and `param2`, you use the syntax:

```
<xsql:action handler="yourpackage.YourCustomHandler" param1="xxx" param2="yyy"/>
```

Some action handlers, perhaps in addition to attributes, may expect text content or element content to appear inside the `<xsql:action>` element. If this is the case, simply use the expected syntax like:

```
<xsql:action handler="yourpackage.YourCustomHandler" param1="xxx" param2="yyy">
   Some Text Goes Here
</xsql:action>
```

or this:

```
<xsql:action handler="yourpackage.YourCustomHandler" param1="xxx" param2="yyy">
  <some>
    <other/>
    <elements/>
    <here/>
  </some>
</xsql:action>
```

Description of XSQL Servlet Examples

Figure 8–14 lists the XSQL Servlet example applications supplied with the software in the `./demo` directory.

Table 8–14 XSQL Servlet Examples

Demonstration Name	Description
Hello World `./demo/helloworld`	Simplest possible XSQL page.
Do You XML Site `./demo/doyouxml`	XSQL page which shows how a simple, data-driven Web site can be built using an XSQL page which makes clever 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 `<xsql:query>` tags, as well as within the attributes to `<xsql:query>` tags to control things like how many records to display and to skip (for "paging" through query results in a stateless way).
Employee Page `./demo/emp`	XSQL page showing XML data from the HR schema's EMPLOYEES table, using XSQL page parameters to control what employees are returned and what column(s) to use for the database sort. Uses an associated XSLT Stylesheet for format the results as an HTML Form containing the `emp.xsql` page as the form action so the user can refine their search criteria.
Insurance Claim Page `./demo/insclaim`	Demonstrates a number of sample queries over the richly-structured, Insurance Claim object view. The `insclaim.sql` sets up the INSURANCE_CLAIM_VIEW object view and populates some sample data.
Invalid Classes Page `./demo/classerr`	XSQL Page which uses invalidclasses.xsl to format a "live" 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.

Table 8–14 (Cont.) XSQL Servlet Examples

Demonstration Name	Description
Airport Code Validation `./demo/airport`	XSQL page returns a "datagram" of information about airports based on their three-letter codes and uses `<xsql:no-rows-query>` 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. `airport.htm` page demonstrates how to use the XML results of `airport.xsql` page from a Web page using JavaScript to exploit built-in XML Document Object Model (DOM) functionality in Internet Explorer 5.0. When you enter the three-letter airport code on the Web page, a JavaScript fetches the XML datagram from XSQL Servlet over the Web corresponding to the code you entered. If the return indicates no match, the program collects a "picklist" of possible matches based on information returned in the XML "datagram" from XSQL Servlet
Airport Code Display `./demo/airport`	Demonstrates using the same XSQL page as the previous example but supplying an XSLT Stylesheet name in the request. This causes the airport information to be formatted as an HTML form instead of being returned as raw XML.
Airport Code Display `./demo/airport`	Demonstrates returning Airport information as a SOAP Service.
Emp/Dept Object Demo `./demo/empdept`	Demonstrates using an object view to group master/detail information from two existing flat tables like EMP and DEPT. The empdeptobjs.sql script creates the object view (along with INSTEAD OF INSERT triggers allowing the master/detail view to be used as an insert target of `xsql:insert-request`). The empdept.xsl stylesheet illustrates an example of the simple form of an XSLT stylesheet that can look just like an HTML page without the extra `xsl:stylesheet` or `xsl:transform` at the top. This is part of the XSLT 1.0 specification called using a Literal Result Element as Stylesheet. It also demonstrates how to generate an HTML page that includes the `<link rel="stylesheet">` to allow the generated HTML to fully leverage CSS for centralized HTML style information, found in the `coolcolors.css` file.
Adhoc Query Visualization `./demo/adhocsql`	Demonstrates passing the entire SQL query and XSLT Stylesheet to use as parameters to the server.
XML Document Demo `./demo/document`	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 `<xsql:include-xsql>` 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.

Table 8–14 (Cont.) XSQL Servlet Examples

Demonstration Name	Description
XML Insert Request Demo `./demo/insertxml`	Demonstrates posting XML from a client to an XSQL Page that handles inserting the posted XML information into a database table using the `<xsql:insert-request>` action element. The demo is setup to accept 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 `insertnewsstory.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 `<article>` element several times within the `<moreovernews>` element to insert several new articles in one shot. The newsstory.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.
SVG Demo `./demo/svg`	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.
PDF Demo `./demo/fop`	`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.

Setting Up the Demo Data

To set up the demo data do the following:

1. Change directory to the `./demo` directory.
2. In this directory, run SQLPLUS. Connect to your database as CTXSYS/CTXSYS — the schema owner for Oracle Text (Intermedia Text) packages — and issue the command

   ```
   GRANT EXECUTE ON ctx_ddl TO scott;
   ```

3. Connect to your database as SYSTEM/MANAGER and issue the command:

   ```
   GRANT QUERY REWRITE TO scott;
   ```

   This allows SCOTT to create a function-based index that one of the demos uses to perform case-insensitive queries on descriptions of airports.
4. Connect to your database as SCOTT/TIGER.
5. Run the script `install.sql` in the `./demo` directory. This script runs all SQL scripts for all the demos.

   ```
   install.sql
   @@insclaim/insclaim.sql
   @@document/docdemo.sql
   @@classerr/invalidclasses.sql
   @@airport/airport.sql
   @@insertxml/newsstory.sql
   @@empdept/empdeptobjs.sql
   ```

6. Change directory to `./doyouxml` subdirectory, and run the following:

   ```
   imp scott/tiger file=doyouxml.dmp
   ```

to import sample data for the "Do You XML? Site" demo.

7. To experience the Scalable Vector Graphics (SVG) demonstration, install an SVG plug-in into your browser, such as Adobe SVG Plug-in.

Advanced XSQL Pages Topics

These sections discuss XSQL Pages advanced topics.

Using a Custom XSQL Configuration File Name

By default, the XSQL Pages framework expects its configuration file to be named `XSQLConfig.xml`. When going between development, test, and production environments, you might want to easily switch between different versions of an XSQL configuration file. To override the name of the configuration file the XSQL page processor will read, do one of the following:

Set the Java system property `xsql.config`. The simplest way is to specify a Java VM command-line flag like `-Dxsql.config=`*`MyConfigFile.xml`* by defining a servlet initialization parameter `xsql.config`

This is accomplished by adding an `<init-param>` element to your `web.xml` file as part of the `<servlet>` tag that defines the XSQL Servlet as follows:

```
    :
 <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>
    :
```

Of course, the servlet initialization parameter is only applicable to the servlet-based use of the XSQL Pages engine. When using the `XSQLCommandLine` or `XSQLRequest` programmatic interfaces, use the System parameter instead.

Note: The config file is always read from the CLASSPATH. For example, if you specify a custom configuration parameter file named `MyConfigFile.xml`, then the XSQL page processor will attempt to read the XML file as a resource from the CLASSPATH. In a J2EE-style servlet environment, that means you must put your `MyConfigFile.xml` into 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, the servlet initialization parameter value is used.

Understanding Client Stylesheet-Override Options

If the current XSQL page being requested allows it, you can supply an XSLT stylesheet URL in the request to override the default stylesheet that is used, or to 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:

- Any relative URL, interpreted relative to the XSQL page being processed
- Any absolute URL using the http protocol scheme, provided it references a trusted host (as defined in the XSQL configuration file, by default named `XSQLConfig.xml`)
- The literal value `none`

This last value, `xml-stylesheet=none`, is particularly useful during development to temporarily "short-circuit" the XSLT stylesheet processing to see what XML datagram your stylesheet is actually seeing. This can help understand why a stylesheet might not be producing the expected results.

Client-override of stylesheets for an XSQL page can be disallowed either by:

- Setting the `allow-client-style` configuration parameter to `no` in the XSQL configuration file, or
- 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 How Stylesheets Are Processed

Here are some points to consider:

Controlling the Content Type of the Returned Document

Setting the content type of the information you serve is very important. It allows the requesting client to correctly interpret the information that you send back.If your stylesheet uses an `<xsl:output>` element, the XSQL Page Processor infers the media type and encoding of the returned document from the `media-type` and `encoding` attributes of `<xsl:output>`.

For example, the following stylesheet uses the `media-type="application/vnd.ms-excel"` attribute on `<xsl:output>` to transform the results of an XSQL page containing a standard query over the HR schema's `employees` table into Microsoft Excel spreadsheet format.

```
<?xml version="1.0"?>
<!-- empToExcel.xsl -->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" media-type="application/vnd.ms-excel"/>
  <xsl:template match="/">
   <html>
     <table>
       <tr><th>Id</th><th>Email</th><th>Salary</th></tr>
       <xsl:for-each select="ROWSET/ROW">
         <tr>
           <td><xsl:value-of select="EMPLOYEE_ID"/></td>
           <td><xsl:value-of select="EMAIL"/></td>
```

```
          <td><xsl:value-of select="SALARY"/></td>
        </tr>
      </xsl:for-each>
    </table>
  </html>
 </xsl:template>
</xsl:stylesheet>
```

An XSQL page that makes use of this stylesheet looks like this:

```
<?xml version="1.0"?>
<?xml-stylesheet href="empToExcel.xsl" type="text/xsl"?>
<xsql:query connection="hr" xmlns:xsql="urn:oracle-xsql">
  select EMPLOYEE_ID, EMAIL, SALARY from employees order by salary desc
</xsql:query>
```

Assigning the Stylesheet Dynamically

As we've seen, if you include an `<?xml-stylesheet?>` processing instruction at the top of your `.xsql` file, it will be considered by the XSQL page processor for use in transforming the resulting XML datagram. For example:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="emp.xsl"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
  <xsql:query>
    SELECT * FROM employees ORDER BY salary DESC
  </xsql:query>
</page>
```

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 stylesheet is accessed by the relative or absolute URL provided in the `href` pseudo-attribute on the `<?xml-stylesheet?>` processing instruction.

By including one or more parameter references in the value of the `href` pseudo-attribute, you can dynamically determine the name of the stylesheet. For example, this page selects the name of the stylesheet to use from a table by assigning the value of a page-private parameter using a query.

```
<?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 Stylesheets in the Client

Some browsers like Microsoft's Internet Explorer 5.0 and higher support processing XSLT stylesheets in the client. These browsers recognize the stylesheet to be processed for an XML document in the same way that a server-side XSQL page does, using an `<?xml-stylesheet?>` processing instruction. This is not a coincidence. 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 page processor performs XSLT transformations in the server, however by adding on additional pseudo-attribute to your `<?xml-stylesheet?>` processing instruction in your XSQL page — `client="yes"` — the page processor will defer the XSLT processing to the client by serving the XML datagram "raw", with the current `<?xml-stylesheet?>` at the top of the document.

One important point to note is that Internet Explorer 5.0 shipped in late 1998, containing an implementation of the XSL stylesheet language that conformed to a December 1998 Working Draft of the standard. The XSLT 1.0 Recommendation that finally emerged in November of 1999 had significant changes from the earlier working draft version on which IE5 is based. This means that IE5 browsers understand a different "dialect" of XSLT than all other XSLT processors — like the Oracle XSLT processor — which implement the XSLT 1.0 Recommendation syntax.

Toward the end of 2000, Microsoft released version 3.0 of their MSXML components as a Web-downloadable release. This latest version *does* implement the XSLT 1.0 standard, however in order for it to be used as the XSLT processor inside the IE5 browser, the user must go through additional installation steps. There is no way for a server to detect that the IE5 browser has installed the latest XSLT components, so until the Internet Explorer 6.0 release emerges, which will contain the latest components by default and which will send a detectable and different User-Agent string containing the 6.0 version number, stylesheets delivered for client processing to IE5 browsers have to use the earlier IE5-"flavor" of XSL.

What we need is a way to request that an XSQL page use different stylesheets depending on the User-Agent making the request. Luckily, the XSQL Pages framework makes this easy and we learn how in the next section.

Providing Multiple, UserAgent-Specific Stylesheets

You can include multiple `<?xml-stylesheet?>` processing instructions at the top of an XSQL page and any of them can contain an optional `media` pseudo-attribute. If specified, the `media` pseudo-attribute's value is compared case-insensitively with the value of the HTTP header's User-Agent string. If the value of the `media` pseudo-attribute matches a part of the User-Agent string, then the processor selects the current `<?xml-stylesheet?>` processing instruction for use, otherwise it ignores it and continues looking. The first matching processing instruction in document order will be used. A processing instruction *without* a `media` pseudo-attribute matches all user agents so it can be used as the fallback/default.

For example, the following processing instructions at the top of an .xsql file...

```
<?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" ?>
<page xmlns:xsql="urn:oracle-xsql" connection="demo">
 ...
```

will use `doyouxml-lynx.xsl` for Lynx browsers, `doyouxml-ie.xsl` for Internet Explorer 5.0 or 5.5 browsers, and `doyouxml.xsl` for all others.

Table 8–15 summarizes all of the supported pseudo-attributes allowed on the `<?xml-stylesheet?>` processing instruction.

Table 8–15 Pseudo-Attributes for <?xml-stylesheet?>

Attribute Name	Description
`type = "string"`	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.
`href = "URL"`	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`).
`media = "string"`	This attribute is optional. If provided, its value is used to perform a case-*insensitive* match on the `User-Agent` string from the HTTP header sent by the requesting device. The current `<?xml-stylesheet?>` processing instruction will only be used if the `User-Agent` string contains the value of the `media` attribute, otherwise it is ignored.
`client = "boolean"`	If set to `yes`, caused the XSQL page processor to defer the processing of the associated XSLT stylesheet to the client. The "raw" XML datagram will be sent to the client with the current `<?xml-stylesheet?>` processing instruction at the top of the document. The default if not specified is to perform the transform in the server.
`serializer = "string"`	By default, the XSQL page processor uses the: ▪ XML DOM serializer if no XSLT stylesheet is used ▪ XSLT processor's serializer, if 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 (by default, named `XSQLConfig.xml`), or the string `java:fully.qualified.Classname`. If both an XSLT stylesheet and the serializer attribute are present, then the XSLT transform is performed first, then the custom serializer is invoked to render the final result to the `OutputStream` or `PrintWriter`.

Using XSQL Configuration File to Tune Your Environment

You can use the XSQL configuration file (by default, named `XSQLConfig.xml`) to tune your XSQL pages environment. Table 8–16 defines all of the parameters that can be set.

Table 8–16 XSQL Configuration File Settings

Configuration Setting Name	Description
`XSQLConfig/servlet/output-buffer-size`	Sets the size (in bytes) of the buffered output stream. If your servlet engine already buffers I/O to the Servlet Output Stream, then you can set to `0` to avoid additional buffering. Default value is `0`. Valid value is any non-negative integer.
`XSQLConfig/servlet/suppress-mime-charset/media-type`	The XSQL Servlet sets the HTTP `ContentType` header to indicate the MIME type of the resource being returned to the request. By default, the XSQL Servlet includes the optional character set information in the MIME type. For a particular MIME type, you can suppress the inclusion of the character set information by including a `<media-type>` element, with the desired MIME type as its contents. You may list any number of `<media-type>` elements. Valid value is any string.
`XSQLConfig/processor/character-set-conversion/default-charset`	By default, the XSQL page processor does character set conversion 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 character set `8859_1` corresponding to IANA's `ISO-8859-1` character set. If your servlet engine uses a different character set as its base character set you can now specify that value here. To suppress character set conversion, specify the empty element `<none/>` as the content of the `<default-charset>` element, instead of a character set name. This is useful if you are working with parameter values that are correctly representable using your servlet default character set, and eliminates a small amount of overhead associated with performing the character set conversion. Valid values are any Java character set name, or the element `<none/>`.

Table 8–16 (Cont.) XSQL Configuration File Settings

Configuration Setting Name	Description
`XSQLConfig/processor/reload-connections-on-error`	Connection definitions are cached when the XSQL Page Processor is initialized. Set this setting to `yes` 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's not in the cached connection list. The `yes` setting is useful during development when you might be adding new `<connection>` 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. Default is `yes`. Valid values are `yes` and `no`.
`XSQLConfig/processor/default-fetch-size`	Sets the default value of the row fetch size for retrieving information from SQL queries from the database. Only takes effect if you are using the Oracle JDBC Driver, otherwise the setting is ignored. Useful for reducing 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.
`XSQLConfig/processor/page-cache-size`	Sets the size of the XSQL cache for XSQL page templates. This determines the maximum number of XSQL pages that will be cached. Least recently used pages get "bumped" out of the cache if you go beyond this number. Default is `25`. Valid value is any nonzero positive integer.
`XSQLConfig/processor/stylesheet-cache-size`	Sets the size of the XSQL cache for XSLT stylesheets. This determines the maximum number of stylesheets that will be cached. Least recently used stylesheets get "bumped" out of the cache if you go beyond this number. Default is `25`. Valid value is any nonzero positive integer.
`XSQLConfig/processor/stylesheet-pool/initial`	Each cached stylesheet is actually 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.

Table 8–16 (Cont.) XSQL Configuration File Settings

Configuration Setting Name	Description
`XSQLConfig/processor/stylesheet-pool/increment`	Sets the number of stylesheets to be allocated when the stylesheet pool must grow due to increased load on the server. Default is `1`. Valid value is any nonzero positive integer.
`XSQLConfig/processor/stylesheet-pool/timeout-seconds`	Sets the number of seconds of inactivity that must transpire before a stylesheet instance in the pool will be 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.
`XSQLConfig/processor/connection-pool/initial`	The XSQL page processor's default connection manager implements connection pooling to improve throughput. This setting controls the initial number of JDBC connections to be allocated in each connection pool. Default is `2`. Valid value is any nonzero positive integer.
`XSQLConfig/processor/connection-pool/increment`	Sets the number of connections to be allocated when the connection pool must grow due to increased load on the server. Default is `1`. Valid value is any nonzero positive integer.
`XSQLConfig/processor/connection-pool/timeout-seconds`	Sets the number of seconds of inactivity that must transpire before a JDBC connection in the pool will be 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.
`XSQLConfig/processor/connection-pool/dump-allowed`	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`.
`XSQLConfig/processor/connection-manager/factory`	Specifies the fully-qualified Java class name of the XSQL connection manager factory implementation. If not specified, this setting defaults to `oracle.xml.xsql.XSQLConnectionManagerFactoryImpl`. Default is `oracle.xml.xsql.XSQLConnectionManagerFactoryImpl`. Valid value is any class name that implements the `oracle.xml.xsql.XSQLConnectionManagerFactory` interface.

Table 8–16 (Cont.) XSQL Configuration File Settings

Configuration Setting Name	Description
`XSQLConfig/processor/owa/fetch-style`	Sets the default OWA Page Buffer fetch style used by the <xsql:include-owa> action.Valid values are `CLOB` or `TABLE`, and the default if not specified is `CLOB`. If set to `CLOB`, the processor uses temporary CLOB to retrieve the OWA page buffer. If set to `TABLE` the processor uses a more efficient approach that requires the existence of the Oracle user-defined type named `XSQL_OWA_ARRAY` which must be created by hand using the DDL statement: `CREATE TYPE xsql_owa_array AS TABLE OF VARCHAR2(32767)`
`XSQLConfig/processor/timing/page`	Determines whether a the XSQL page processor adds an `xsql-timing` attribute to the document element of the page whose value reports the elapsed number of milliseconds required to process the page. Default is `no`. Valid value is `yes` or `no`.
`XSQLConfig/processor/timing/action`	Determines whether a 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. Default is `no`. Valid value is `yes` or `no`.
`XSQLConfig/processor/logger/factory`	Specifies the fully-qualified Java class name of a custom XSQL logger factory implementation. If not specified, then no logger is used. Valid value is any class name that implements the `oracle.xml.xsql.XSQLLoggerFactory` interface.
`XSQLConfig/processor/error-handler/class`	Specifies the fully-qualified Java class name of a custom XSQL error handler to be the default error handler implementation. If not specified, then the default error handler is used. Valid value is any class name that implements the `oracle.xml.xsql.XSQLErrorHandler` interface.

Table 8–16 (Cont.) XSQL Configuration File Settings

Configuration Setting Name	Description
`XSQLConfig/processor/xml-parsing/preserve-whitespace`	Determines whether the XSQL page processor preserves whitespace when parsing XSQL page templates and XSLT stylesheets. The default value is `true`. Valid values are `true` or `false`. Changing the default to false can slightly speed up the processing of XSQL pages and stylesheets since ignoring whitespace while parsing is faster than preserving it.
`XSQLConfig/processor/security/stylesheet/defaults/allow-client-style`	While developing an application, it is frequently useful to take advantage of the XSQL page processor's for each request stylesheet override capability by providing a value for the special `xml-stylesheet` parameter in the request. One of the most common uses is to provide the `xml-stylesheet=none` combination to temporarily disable the application of the stylesheet to "peek" underneath at the raw XSQL data page for debugging purposes. When development is completed, you can explicitly add the `allow-client-style="no"` attribute to the document element of each XSQL page to prohibit client overriding of the stylesheet in the production application. However, using this configuration setting, you can globally change the default behavior for `allow-client-style` in a single place. Note that this only provides the *default* setting for this behavior. If the `allow-client-style="yes|no"` attribute is explicitly specified on the document element for a given XSQL page, its value takes precedence over this global default. Valid values are `yes` and `no`.

Table 8–16 (Cont.) XSQL Configuration File Settings

Configuration Setting Name	Description
`XSQLConfig/processor/security/stylesheet/trusted-hosts/host`	XSLT stylesheets can invoke extension functions. In particular, the Oracle XSLT processor — which the XSQL page processor uses to process all XSLT stylesheets — supports *Java* extension functions. Typically your XSQL pages will refer to XSLT stylesheets using relative URL's The XSQL page processor enforces that any absolute URL to an XSLT stylesheet that is processed must be from a trusted host whose name is listed here in the configuration file. You may list any number of `<host>` elements inside the `<trusted-hosts>` element. The name of the local machine, `localhost`, and `127.0.0.1` are considered trusted hosts by default. Valid values are any hostname or IP address.
`XSQLConfig/http/proxyhost`	Sets the name of the HTTP proxy server to use when processing `URL`s with the http protocol scheme. Valid value is any hostname or IP address.
`XSQLConfig/http/proxyport`	Sets the port number of the HTTP proxy server to use when processing `URL`s with the http protocol scheme. Valid value is any nonzero integer.
`XSQLConfig/connectiondefs/connection`	Defines a "nickname" and the JDBC connection details for a named connection for use by the XSQL page processor. You may supply any number of `<connection>` element children of `<connectiondefs>`. Each connection definition must supply a `name` attribute, and may supply appropriate children elements `<username>, <password>, <driver>, <dburl>`, and `<autocommit>`.
`XSQLConfig/connectiondefs/connection/username`	Defines the username for the current connection.
`XSQLConfig/connectiondefs/connection/password`	Defines the password for the current connection.
`XSQLConfig/connectiondefs/connection/dburl`	Defines the JDBC connection URL for the current connection.
`XSQLConfig/connectiondefs/connection/driver`	Specifies the fully-qualified Java class name of the JDBC driver to be used for the current connection. If not specified, defaults to `oracle.jdbc.driver.OracleDriver`.

Table 8–16 (Cont.) XSQL Configuration File Settings

Configuration Setting Name	Description
`XSQLConfig/connectiondefs/connection/autocommit`	Explicitly sets the Auto Commit flag for the current connection. If not specified, connection uses JDBC driver's default setting for Auto Commit.
`XSQLConfig/serializerdefs/serializer`	Defines a named custom serializer implementation. You may supply any number of `<serializer>` element children of `<serializerdefs>`. Each must specify both a `<name>` and a `<class>` child element.
`XSQLConfig/serializerdefs/serializer/name`	Defines the name of the current custom serializer definition.
`XSQLConfig/connectiondefs/connection/class`	Specifies the fully-qualified Java class name of the current custom serializer. The class must implement the `oracle.xml.xsql.XSQLDocumentSerializer` interface.

Using the FOP Serializer to Produce PDF Output

Using the XSQL Pages framework's support for custom serializers, the `oracle.xml.xsql.serializers.XSQLFOPSerializer` is provided for integrating with the Apache FOP processor (http://xml.apache.org/fop). The FOP processor renders a PDF document from an XML document containing XSL Formatting Objects (`http://www.w3.org/TR/xsl`).

For example, given the following XSLT stylesheet, `EmpTableFO.xsl`:

```
<?xml version="1.0"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format" xsl:version="1.0"
         xmlns:xsl="http://www.w3.org/1999/XSL/Transform">


  <!-- defines the layout master -->
  <fo:layout-master-set>
    <fo:simple-page-master master-name="first"
                           page-height="29.7cm"
                           page-width="21cm"
                           margin-top="1cm"
                           margin-bottom="2cm"
                           margin-left="2.5cm"
                           margin-right="2.5cm">
      <fo:region-body margin-top="3cm"/>
    </fo:simple-page-master>
  </fo:layout-master-set>

  <!-- starts actual layout -->
  <fo:page-sequence master-reference="first">

  <fo:flow flow-name="xsl-region-body">

      <fo:block font-size="24pt" line-height="24pt" font-weight="bold"
start-indent="15pt">
        Total of All Salaries is $<xsl:value-of select="sum(/ROWSET/ROW/SAL)"/>
      </fo:block>

      <!-- Here starts the table -->
      <fo:block border-width="2pt">
        <fo:table>
          <fo:table-column column-width="4cm"/>
          <fo:table-column column-width="4cm"/>
          <fo:table-body font-size="10pt" font-family="sans-serif">
            <xsl:for-each select="ROWSET/ROW">
              <fo:table-row line-height="12pt">
                <fo:table-cell>
                  <fo:block><xsl:value-of select="ENAME"/></fo:block>
                </fo:table-cell>
                <fo:table-cell>
                  <fo:block><xsl:value-of select="SAL"/></fo:block>
                </fo:table-cell>
              </fo:table-row>
            </xsl:for-each>
          </fo:table-body>
        </fo:table>
      </fo:block>
    </fo:flow>
  </fo:page-sequence>
</fo:root>
```

Note: To use the XSQL FOP Serializer, you need to add these additional Java archives to your server's CLASSPATH:

- `xsqlserializers.jar` - supplied with Oracle XSQL
- 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

For reference, in case you might want to customize the implementation, the source code for the FOP Serializer provided in this release looks like this:

```
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);
    }
  }
}
```

Using XSQL Page Processor Programmatically

The `XSQLRequest` class, enables you to utilize the XSQL page processor "engine" from within your own custom Java programs. Using the API is simple. You 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`

Then you invoke one of the following methods to process the page:

- `process()` to write the result to a `PrintWriter` or `OutputStream`, or
- `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, if you want to use your own connection manager implementation.

Note that the ability to pass the XSQL page to be processed as an in-memory XML Document object means that you can dynamically generate any valid XSQL page for processing using any means necessary, then pass the page to the XSQL engine for evaluation.

When processing a page, there are two additional things you may want to do as part of the request:

- Pass a set of parameters to the request

  You accomplish this 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 using the `setPostedDocument()` method on the XSQLRequest object.

Here is a simple example of processing a page using `XSQLRequest`:

```
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);
   // Setup 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));
  }
}
```

Writing Custom XSQL Action Handlers

When the task at hand requires custom processing, and none of the built-in actions does exactly what you need, you can augment your repertoire by writing your own actions that any of your XSQL pages can use.

The XSQL page processor at its very core is an engine that processes XML documents containing "action elements". The page processor engine is written to support any action that implements the `XSQLActionHandler` interface. All of the built-in actions implement this interface.

The XSQL Page Processor processes the actions in a page in the following way. For each action in the page, the engine:

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 knows the mapping of XSQL action element name to the Java class that implements the action's handler. Table 8–17, " Built-In XSQL Elements and Action Handler Classes" lists that mapping explicitly for your reference. For user-defined actions, you use the built-in:

```
<xsql:action handler="fully.qualified.Classname" ... />
```

action whose `handler` attribute provides the fully-qualified name of the Java class that implements the custom action handler.

Table 8–17 Built-In XSQL Elements and Action Handler Classes

XSQL Action Element	Handler Class in oracle.xml.xsql.actions	Description
`<xsql:query>`	`XSQLQueryHandler`	Execute an arbitrary SQL statement and include its result in canonical XML format.
`<xsql:dml>`	`XSQLDMLHandler`	Execute a SQL DML statement or a PL/SQL anonymous block.
`<xsql:set-stylesheet-param>`	`XSQLStylesheetParameterHandler`	Set the value of a top-level XSLT stylesheet parameter.
`<xsql:insert-request>`	`XSQLInsertRequestHandler`	Insert the XML document (or HTML form) posted in the request into a database table or view.
`<xsql:include-xml>`	`XSQLIncludeXMLHandler`	Include arbitrary XML resources at any point in your page by relative or absolute URL.
`<xsql:include-request-params>`	`XSQLIncludeRequestHandler`	Include all request parameters as XML elements in your XSQL page.
`<xsql:include-posted-xml>`	`XSQLIncludePostedXMLHandler`	
`<xsql:include-xsql>`	`XSQLIncludeXSQLHandler`	Include the results of one XSQL page at any point inside another.
`<xsql:include-owa>`	`XSQLIncludeOWAHandler`	Include the results of executing a stored procedure that makes use of the Oracle Web Agent (OWA) packages inside the database to generate XML.
`<xsql:action>`	`XSQLExtensionActionHandler`	Invoke a user-defined action handler, implemented in Java, for executing custom logic and including custom XML information into your XSQL page.
`<xsql:ref-cursor-function>`	`XSQLRefCursorFunctionHandler`	Includes the canonical XML representation of the result set of a cursor returned by a PL/SQL stored function.
`<xsql:include-param>`	`XSQLGetParameterHandler`	Include a parameter and its value as an element in your XSQL page.
`<xsql:if-param>`	`XSQLIfParamHandler`	Conditionally include XML content and other XSQL actions (or other XSQL actions).

Table 8–17 (Cont.) Built-In XSQL Elements and Action Handler Classes

XSQL Action Element	Handler Class in oracle.xml.xsql.actions	Description
`<xsql:set-session-param>`	`XSQLSetSessionParamHandler`	Set an HTTP-Session level parameter.
`<xsql:set-page-param>`	`XSQLSetPageParamHandler`	Set an HTTP-Session level parameter. Set a page-level (local) parameter that can be referred to in subsequent SQL statements in the page.
`<xsql:set-cookie>`	`XSQLSetCookieHandler`	Set an HTTP Cookie.
`<xsql:insert-param>`	`XSQLInsertParameterHandler`	Inserts the XML document contained in the value of a single parameter.
`<xsql:update-request>`	`XSQLUpdateRequestHandler`	Update an existing row in the database based on the posted XML document supplied in the request.
`<xsql:delete-request>`	`XSQLDeleteRequestHandler`	Delete an existing row in the database based on the posted XML document supplied in the request.
`<xsql:if-param>`		Includes nested actions and literal XML content (or literal XML content) if some condition based on a parameter value is true.

All the demos are listed at `http://localhost/xsql/index.html`.

Writing your Own Action Handler

To create a custom Action Handler, you need to provide a class that implements the `oracle.xml.xsql.XSQLActionHandler` interface. Most custom action handlers extend `oracle.xml.xsql.XSQLActionHandlerImpl` that provides a default implementation of the `init()` method and offers a set of useful helper methods that will prove very useful.

When an action handler's handleAction method is invoked by the XSQL page processor, the action implementation gets passed the root node of a DOM Document Fragment to which the action handler appends any dynamically created XML content that is returned to the page.

The XSQL Page Processor conceptually replaces the action element in the XSQL page template with the content of this Document Fragment. It is completely legal for an Action Handler to append nothing to this document fragment, if it has no XML content to add to the page.

While writing you custom action handlers, several methods on the `XSQLActionHandlerImpl` class are worth noting because they make your life a lot easier. Table 8–18 lists the methods that will likely come in handy for you.

Table 8–18 Helpful Methods on oracle.xml.xsql.SQLActionHandlerImpl

Method Name	Description
`getActionElement`	Returns the current action element being handled
`getActionElementContent`	Returns the text content of the current action element, with all lexical parameters substituted appropriately.
`getPageRequest`	Returns the current XSQL page processor context. Using this object you can then do things like: ■ `setPageParam()` Set a page parameter value ■ `getPostedDocument()/setPostedDocument()` Get or set the posted XML document ■ `translateURL()` Translate a relative URL to an absolute URL ■ `getRequestObject()/setRequestObject()` Get or set objects in the page request context that can be shared across actions in a single page. ■ `getJDBCConnection()` Gets the JDBC connection in use by this page (possible null if no connection in use). ■ `getRequestType()` 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 addition Servlet-specific methods like `getHttpServletRequest`, `getHttpServletResponse`, and `getServletContext`
`getAttributeAllowingParam`	Retrieve the attribute value from an element, resolving any XSQL lexical parameter references that might appear in the attribute's value. Typically this method is applied to the action element itself, but it is also useful for accessing attributes of any of its sub-elements. To access an attribute value without allowing lexical parameters, use the standard `getAttribute()` method on the DOM Element interface.
`appendSecondaryDocument`	Append the entire contents of an external XML document to the root of the action handler result content.
`addResultElement`	Simplify appending a single element with text content to the root of the action handler result content.
`firstColumnOfFirstRow`	Return the first column value of the first row of a SQL statement passed in. Requires the current page to have a connection attribute on its document element, or an error is returned.
`bindVariableCount`	Returns the number of tokens in the space-delimited list of `bind-params`, indicating how many bind variables are expected to be bound to parameters.

Table 8–18 (Cont.) Helpful Methods on oracle.xml.xsql.SQLActionHandlerImpl

Method Name	Description
`handleBindVariables`	Manage the binding of JDBC bind variables that appear in a prepared statement with the parameter values specified in the `bind-params` attribute on the current action element. If the statement already is using a number of bind variables prior to call this method, you can pass the number of existing bind variable "slots" in use as well.
`reportErrorIncludingStatement`	Report an error, including the offending (SQL) statement that caused the problem, optionally including a numeric error code.
`reportFatalError`	Report a fatal error.
`reportMissingAttribute`	Report an error that a required action handler attribute is missing using the standard `<xsql-error>` element.
`reportStatus`	Report action handler status using the standard `<xsql-status>` element.
`requiredConnectionProvided`	Checks whether a connection is available for this request, and outputs an "errorgram" into the page if no connection is available.
`variableValue`	Returns the value of a lexical parameter, taking into account all scoping rules which might determine its default value.

The following example shows a custom action handler `MyIncludeXSQLHandler` that leverages one of the built-in action handlers and then uses arbitrary Java code to modify the resulting XML fragment returned by that handler before appending its result to the XSQL page:

```
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);
    }
}
```

If you create custom action handlers that need to work differently based on whether the page is being requested through the XSQL Servlet, the XSQL Command-line Utility, or programmatically through the XSQLRequest class, then in your Action Handler implementation you can call `getPageRequest()` to get a reference to the XSQLPageRequest interface for the current page request. By calling `getRequestType()` on the XSQLPageRequest object, you can see if the request is coming from the "Servlet", "Command Line", or "Programmatic" routes respectively. If the return value is "Servlet", then you can get access to the HTTP Servlet request, response, and servlet context objects by doing:

```
XSQLServletPageRequest xspr = (XSQLServletPageRequest)getPageRequest();
if (xspr.getRequestType().equals("Servlet")) {
  HttpServletRequest     req  = xspr.getHttpServletRequest();
  HttpServletResponse   resp  = xspr.getHttpServletResponse();
  ServletContext        cont  = xspr.getServletContext();
  // do something fun here with req, resp, or cont however
  // writing to the response directly from a handler will
  // produce unexpected results. Allow the XSQL Servlet
  // or your custom Serializer to write to the servlet
  // response output stream at the write moment later when all
  // action elements have been processed.
}
```

Using Multi-Valued Parameters in Custom XSQL Actions

The base class for custom XSQL actions, `XSQLActionHandlerImpl` supports working with array-named lexical parameter substitution and array-named bind variables as well as simple-valued parameters. If your custom actions are use methods like `getAttributeAllowingParam()`,`getActionElementContent()`, or `handleBindVariables()` methods from this base class, you pickup the 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` makes it easy to use this functionality from within a custom action handler if you need to do so programmatically.

Writing Custom XSQL Serializers

You can provide a user-defined serializer class to programmatically control how the final XSQL datapage's XML document is serialized to a text or binary stream. A user-defined serializer must implement the `oracle.xml.xsql.XSQLDocumentSerializer` interface which comprises the single method:

```
void serialize(org.w3c.dom.Document doc, XSQLPageRequest env) throws Throwable;
```

In this release, DOM-based serializers are supported. A future release may support SAX2-based serializers as well. 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`).

   You set the type by calling `setContentType()` on the `XSQLPageRequest` that is passed to your serializer. When setting the content type, you can either set just a MIME type like this:

```
env.setContentType("text/html");
```

or a MIME type with an explicit output encoding character set like this:

```
env.setContentType("text/html;charset=Shift_JIS");
```

2. Call `getWriter()` or `getOutputStream()` — but not both! — on the `XSQLPageRequest` to get the appropriate `PrintWriter` or `OutputStream` respectively to use for serializing the content.

For example, the following custom serializer illustrates a simple implementation which simply serializes an HTML document containing the name of the document element of the current XSQL data page:

```
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>");
  }
}
```

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, simply add the `serializer="java:fully.qualified.ClassName"` in the `<?xml-stylesheet?>` processing instruction at the top of your page like this:

```
<?xml version="1.0?>
<?xml-stylesheet type="text/xsl" href="mystyle.xsl"
                 serializer="java:my.pkg.MySerializer"?>
```

If you only need the custom serializer, simply leave out the `type` and `href` attributes like this:

```
<?xml version="1.0?>
<?xml-stylesheet serializer="java:my.pkg.MySerializer"?>
```

You can also assign a short name to your custom serializers in the `<serializerdefs>` section of the XSQL configuration file (by default, named `XSQLConfig.xml`) and then use the nickname (case-sensitive) in the serializer attribute instead to save typing. For example, if you have the following in the XSQL configuration file:

```
<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>
```

then you can use the nicknames "Sample" and "FOP" (or "FOP") as shown in the following examples:

```
<?xml-stylesheet type="text/xsl" href="emp-to-xslfo.xsl" serializer="FOP"?>
```

or

```
<?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 (like a dynamically produced GIF image, for example) can be serialized. Using the XSQL Servlet, writing to this output stream results in writing the binary information to the servlet output stream.

For example, the following serializer illustrates an example of writing out a dynamic GIF image. In this example the GIF image is a static little "ok" icon, but it shows the basic technique that a more sophisticated image serializer needs to use:

```
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)0xFFFFFF80,(byte)0x0,
      (byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,
      (byte)0xFFFFFFFF,(byte)0xFFFFFFFF,(byte)0xFFFFFFFF,(byte)0x2C,
      (byte)0x0,(byte)0x0,(byte)0x0,(byte)0x0,
      (byte)0xB,(byte)0x0,(byte)0x9,(byte)0x0,
      (byte)0x0,(byte)0x2,(byte)0x14,(byte)0xFFFFFF8C,
      (byte)0xF,(byte)0xFFFFFFA7,(byte)0xFFFFFFB8,(byte)0xFFFFFF9B,
      (byte)0xA,(byte)0xFFFFFFA2,(byte)0x79,(byte)0xFFFFFFE9,
      (byte)0xFFFFFF85,(byte)0x7A,(byte)0x27,(byte)0xFFFFFF93,
      (byte)0x5A,(byte)0xFFFFFFE3,(byte)0xFFFFFFEC,(byte)0x75,
      (byte)0x11,(byte)0xFFFFFF85,(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 programmatic 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 will raise 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 may use one of the two provided XSQLConnectionManager implementations that let you use your servlet container's JDBC Datasource implementation and related connection pooling features.

This XSQL Pages release comes with two of these alternative connection manager implementations:

- `oracle.xml.xsql.XSQLDatasourceConnectionManager`

  Consider using this alternative connection manager if your servlet container's datasource implementation does *not* use the Oracle JDBC driver under the covers. Certain features of the XSQL Pages system will not be available when you are not using an Oracle JDBC driver, like `<xsql:ref-cursor-function>` and `<xsql:include-owa>`.

- oracle.xml.xsql.XSQLOracleDatasourceConnectionManager

  Consider using this alternative connection manager when you know that your datasource implementation returns JDBC `PreparedStatement` and `CallableStatement` objects that implement the `oracle.jdbc.PreparedStatement` and `oracle.jdbc.CallableStatement` interfaces respectively. The Oracle Application Server has a datasource implementation that does this.

When using either of these alternative connection manager implementations, the value of the connection attribute in your XSQL page template is the JNDI name used to lookup your desired datasource. For example, the value of the connection attribute might look something like:

- `jdbc/scottDS`
- `java:comp/env/jdbc/MyDatasource`

Remember that if you are not using the default XSQL Pages connection manager, then any connection pooling functionality that you need must be provided by the alternative connection manager implementation. In the case of the earlier two options that are based on JDBC Datasources, you are relying on properly configuring your servlet container to supply the connection pooling. See your servlet container's 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 provide:

1. A connection manager *factory* object that implements the `oracle.xml.xsql.XSQLConnectionManagerFactory` interface.
2. A connection manager object that implements the `oracle.xml.xsql.XSQLConnectionManager` interface.

Your custom connection manager factory can be set to be used as the default connection manager factory by providing the class name in the XSQL configuration file (by default, named `XSQLConfig.xml`) in the section:

```
<!--
     | 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, a custom connection factory can be associated with any individual `XSQLRequest` object 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 like a servlet engine, it is the responsibility of the `XSQLConnectionManager` object to insure that a single `XSQLConnection` instance is not used by two different threads. This can be assured by marking the connection as "in use" for the span of time between the invocation of the `getConnection()` method and the `releaseConnection()` method. 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 as well, then your connection manager will be given a chance to cleanup 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, this will give the connection manager a chance to clean up resources as part of the servlet destruction process.

Providing a Custom XSQLErrorHandler Implementation

You may want to control how serious page processor errors (like a connection's being unavailable) are reported to users. Writing a class that implements the `oracle.xml.xsql.XSQLErrorHandler` interface enables you to do this. The interface contains the single method:

```
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 page processor writes out any page processor error messages. The new `XSQLError` object encapsulates the error information and provides access to the error code, formatted error message, and so on.

For example, here is a sample implementation of `XSQLErrorHandler`:

```
package example;
import oracle.xml.xsql.*;
import java.io.*;
/**
 * Example of a custom XSQLErrorHandler implementation
 */
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 gets used in two distinct ways:

1. You can define the name of a custom `XSQLErrorHandler` implementation class in the XSQL configuration file (by default, named `XSQLConfig.xml`) by providing the fully-qualified class name of your error handler class as the value of the `/XSQLConfig/processor/error-handler/class` entry.
2. If the Page Processor can load this class and it correctly implements the `XSQLErrorHandler` interface, then this class is used as a singleton and replaces the default implementation globally, wherever page processor errors are reported.
3. You can override the error writer on a for each page basis using the new, optional `errorHandler` (or `xsql:errorHandler`) attribute on the document element of your page.
4. The value of this attribute is the fully-qualified class name of a class that implements the `XSQLErrorHandler` interface. This class will be used to report the errors for just this page and the class is instantiated on each page request by the page engine.

You can use a combination of both 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 two interfaces `oracle.xml.xsql.XSQLLoggerFactory` and `oracle.xml.xsql.XSQLLogger`.

The `XSQLLoggerFactory` interface contains the single method:

```
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 Page processor holds a reference to the `XSQLLogger` object returned by the factory for the duration of a page request and uses it to log the start and end of each page request by invoking the `logRequestStart()` and `logRequestEnd()` methods on it.

The `XSQLLogger` interface looks like this:

```
public interface XSQLLogger {
   public void logRequestStart(XSQLPageRequest env) ;
   public void logRequestEnd(XSQLPageRequest env);
}
```

The following two classes illustrate a trivial implementation of a custom logger. First is the `XSQLLogger` implementation which notes the time the page request started and then logs the page request end by printing the name of the page request and the elapsed time to `System.out`:

```
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");
  }
}
```

Next, the factory implementation:

```
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 like this:

```
<XSQLConfig>
    :
  <processor>
         :
      <logger>
         <factory>example.SampleCustomLoggerFactory</factory>
      </logger>
         :
   </processor>
</XSQLConfig>
```

By default, this `<logger>` section is commented out, and there is no default logger.

Formatting XSQL Action Handler Errors

Errors raised by the processing of any XSQL Action Elements are reported as XML elements in a uniform way so that XSL Stylesheets can detect their presence and optionally format them for presentation.

The action element in error will be replaced in the page by:

```
<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

Displaying Error Information on Screen

Here is an example of an XSLT stylesheet that uses this information to display error information on the screen:

```
<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>
```

XSQL Servlet Limitations and Hints

XSQL Servlet has the following limitations:

HTTP Parameters with Multibyte Names

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 using `<xsql:include-request-params>`. An attempt to refer to a parameter with a multibyte name inside the query statement of an `<xsql:query>` tag will return an empty string for the parameter's value.

As a workaround use a non-multibyte parameter name. The parameter can still have a multibyte value which can be handled correctly.

CURSOR() Function in SQL Statements

If you use the CURSOR() function in SQL statements you may get an "Exhausted ResultSet" error if the CURSOR() statements are nested and if the first row of the query returns an empty result set for its CURSOR() function.

Hints for Using the XSQL Servlet

This section lists XSQL Servlet hints.

Specifying a DTD While Transforming XSQL Output to a WML Document

There is a way to specify a particular DTD while transforming XSQL output to a WML document for a wireless application.

The way you do it is using a built-in facility of the XSLT stylesheet called `<xsl:output>`. Here is an example:

```
<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>
```

This will produce an XML result with the following code in it:

```
<!DOCTYPE xxxx SYSTEM "your.dtd">
```

where `"your.dtd"` can be any valid absolute or relative URL.

Writing XSQL Servlet Conditional Statements

It is possible to write conditional statements in an XSQL file such as:

```
<xsql:choose>
    <xsql:when test="@security='admin'">
      <xsql:query>
          SELECT ....
      </xsql:query>
   </xsq:when>
   <xsql:when test="@security='user'">
      <xsql:query>
          SELECT ....
      </xsql:query>
   </xsql:when>
</xsql:if>
```

Use `<xsql:ref-cursor-function>` to call a PL/SQL procedure that conditionally returns a `REF CURSOR` to the appropriate query.

Using a Value Retrieved in One Query in Another Query's Where Clause

If you have two queries in an XSQL file, you can use the value of a select list item of the first query in the second query, using page parameters:

```
<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 ...
  </xsl:set-param-param>
  <xsql:query bind-params="xxx">
     select col3,col4 from table2
    where col3 = ?
  </xsql:query>
</page>
```

Using the XSQL Servlet with Non-Oracle Databases

The XSQL Servlet can connect to any database that has JDBC support. Just indicate the appropriate JDBC driver class and connection URL in the XSQL configuration file (by default, named `XSQLConfig.xml`) connection definition. Of course, object/relational functionality only works when using Oracle with the Oracle JDBC driver.

Handling Multi-Valued HTML Form Parameters

There is a way to handle multi-valued HTML `<form>` parameters which 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.

For example, if you have a multi-valued parameter named `guy`, then you can use the array-parameter notation in an XSQL page like this:

```
<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[]}"
                       treat-list-as-array="yes" quote-array-values="yes"/>
  <xsql:include-param name="guy-list"/>
  <xsql:include-param name="quoted-guys"/>
  <xsql:include-param name="guy[]"/>
</page>
```

If this page is requested with the URL following, containing multiple parameters of the same name to produce a multi-valued attribute:

```
http://yourserver.com/page.xsql?guy=Curly&guy=Larry&guy=Moe
```

then the page returned will be:

```
<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 the multi-valued page parameter preceding nonzero in a SQL statement by using the following code:

```
<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>
    SELECT * FROM sometable WHERE name IN ({@quoted-guys})
  </xsql:query>
</page>
```

Running the XSQL Servlet with Oracle Release 7.3

Make sure you're using the JDBC driver, which can connect to an Oracle release 7.3 database with no problems.

Out Variable is not Supported in <xsql:dml>

You cannot set parameter values by binding them in the position of `OUT` variables in this release using `<xsql:dml>`. Only `IN` parameters are supported for binding. You can create a wrapper procedure that constructs XML elements using the HTTP package and then your XSQL page can invoke the wrapper procedure using `<xsql:include-owa>` instead.

For an example, suppose you had the following procedure:

```
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 following procedure to wrap it, taking all of the `IN` arguments that the procedure preceding expects, and then encoding the `OUT` values as a little XML datagram that you print to the OWA page buffer:

```
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;
```

This way, you can build an XSQL page like this that calls the wrapper procedure:

```
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
  <xsql:include-owa bind-params="arg1 arg2">
    BEGIN addmultwrapper(?,?); END;
  </xsql:include-owa>
</page>
```

This allows a request like the following:

```
http://yourserver.com/addmult.xsql?arg1=30&arg2=45
```

to return an XML datagram that reflects the `OUT` values like this:

```
<page>
  <addmult><sum>75</sum><product>1350</product></addmult>
</page>
```

Receiving "Unable to Connect" Errors

Suppose that you are unable to connect to a database and get errors running a program like the `helloworld.xsql` sample:

```
Oracle XSQL Servlet Page Processor 9.0.0.0.0 (Beta)
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))))
```

If you get this far, it's actually 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 this:

```
<connection name="demo">
  <username>scott</username>
  <password>tiger</password>
  <dburl>jdbc:oracle:thin:@localhost:1521:ORCL</dburl>
  <driver>oracle.jdbc.driver.OracleDriver</driver>
</connection>
```

The error you're getting is likely because of one of the following reasons:

1. Your database is not on the `localhost` machine.
2. Your database `SID` is not `ORCL.`
3. Your TNS Listener Port is not 1521.

Make sure those values are appropriate for your database and you have no problems.

Using Other File Extensions Besides .xsql

The `.xsql` extension is just the default extension used to recognize XSQL pages. You can modify your servlet engine's configuration settings to associate any extension you like with the `oracle.xml.xsql.XSQLServlet` servlet class using the same technique that was used to associate the `*.xsql` extension with it.

Receiving "No Posted Document to Process" when you Try to Post XML

When trying to post XML information to an XSQL page for processing, it must be sent by the HTTP POST method. This can be an HTTP POST-ed HTML Form or an XML document sent by HTTP POST. If you try to use HTTP GET instead, there is no posted document, and hence you get this error. Use HTTP POST instead to have the correct behavior.

XSQL Supports SOAP

Your page can access contents of the inbound SOAP message using the `<xsql:set-page-param>` action's xpath="*`XpathExpression`*" attribute. Alternatively, your customer action handlers can gain direct access to the posted SOAP message body by calling `getPageRequest().getPostedDocument()`. To create the SOAP response body to return to the client, you can either use an XSLT stylesheet or a custom serializer implementation to write out the XML response in an appropriate SOAP-encoded format.

See the supplied AirportSOAP demo that comes with the XSQL Pages framework for an example of using an XSQL page to implement a SOAP-based Web Service.

Passing the Connection for XSQL

Reference an XSQL parameter in your page's `connection` attribute, making sure to define an attribute of the same name to serve as the default value for the connection name. For example:

```
<xsql:query conn="testdb" connection="{@conn}" xmlns:xsql="urn:oracle-xsql">
  ...
</xsql:query>
```

If you retrieve this page without any parameters, the value of the `conn` parameter will be `testdb`, so the page will use the connection named `testdb` defined in the XSQL configuration file (by default, named `XSQLConfig.xml`). If instead you request the page with conn=proddb, then the page will use the connection named proddb instead.

Controlling How Database Connections and Passwords Are Stored

If you need a more sophisticated set of username and password management than the one that is provided by default in XSQL using the XSQL configuration file.

You can completely redefine the way the XSQL Page Processor handles database connections by creating your own implementation of the XSQLConnectionManager interface. To achieve this, you need to write a class that implements the oracle.xml.xsql.XSQLConnectionManagerFactory interface and a class that implements the oracle.xml.xsql.XSQLConnectionManager interface, then change the name of the XSQLConnectionManagerFactory class to use in your XSQL configuration file. Once you've done this, your connection management scheme will be used instead of the XSQL Pages default scheme.

Accessing Authentication Information in a Custom Connection Manager

If you want to use the HTTP authentication mechanism to get the username and password to connect to the database. It is possible to get this kind of information in a custom connection manager's getConnection() method.

The getConnection() method is passed an instance of the XSQLPageRequest interface. From it, you can get the HTTP Request object by:

1. Testing the request type to make sure it's "Servlet"
2. Casting XSQLPageRequest to XSQLServletPageRequest
3. Calling getHttpServletRequest() on the result of (2)

You can then get the authentication information from that HTTP Request object.

Retrieving the Name of the Current XSQL Page

There is a way for an XSQL page to 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 this to retrieve the name of the page inside a custom action handler:

```
// Get the name of the current page from the current page's URI
  private String curPageName(XSQLPageRequest req) {
    String thisPage = req.getSourceDocumentURI();;
    int pos = thisPage.lastIndexOf('/');
    if (pos >=0) thisPage = thisPage.substring(pos+1);
    pos = thisPage.indexOf('?');
    if (pos >=0) thisPage = thisPage.substring(0,pos-1);
    return thisPage;
  }
```

Resolving Errors When Using the FOP Serializer

You can format XML into PDF using Formatting Object (FOP). If you get an error trying to use the FOP Serializer, typically the problem is that you do not have all of the required JAR files in the CLASSPATH. The XSQLFOPSerializer class resides in the separate xsqlserializers.jar file, and this must be in the CLASSPATH to use the FOP integration.

Then, the XSQLFOPSerializer class itself has dependencies on several libraries from Apache. For example, here is the source code for a FOP Serializer that works with the Apache FOP 0.20.3RC release candidate of the FOP software:

```
package sample;
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 SampleFOPSerializer 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);
    }
  }
}
```

This FOP serializer depends on having the following additional Apache JAR files in the CLASSPATH at runtime:

1. `fop.jar` - Apache FOP Rendering Engine
2. `batik.jar` - Apache Batik SVG Rendering Engine
3. `avalon-framework-4.0.jar` - APIs for Apache Avalon Framework
4. `logkit-1.0.jar` - APIs for the Apache Logkit

See Also:

- http://xml.apache.org/fop/
- http://www.xml.com/pub/rg/75

Tuning XSQL Pages for Fastest Performance

The biggest thing that affects the performance is the size of the data you are querying (and of course the pure speed of the queries). Assuming you have tuned your queries and used bind variables instead of lexical bind variables wherever allowed by SQL, then the key remaining tip is to make sure you are only querying the minimum amount of data needed to render the needed result.

If you are querying thousands of rows of data, only to use your XSLT stylesheet to filter the rows to present only 10 of those rows in the browser, then this is a bad choice. Use the database's capabilities to the maximum to filter the rows and return only the 10 rows you care about if at all possible. Think of XSQL as a thin coordination layer between Oracle database and the power of XSLT as a transformation language.

Using XSQL with Other Connection Pool Implementations

You can set up XSQL pages to use connections taken from a connection pool, if for example, you are running XSQL servlet in a WebLogic Web server.

XSQL implements it's own connection pooling so in general you don't have to use another connection pool, but if providing the JDBC connection string of appropriate format is not enough to use the WebLogic pool, then you can create your own custom connection manager for XSQL by implementing the interfaces `XSQLConnectionManagerFactory` and `XSQLConnectionManager`.

Including XML Documents Stored in CLOBs in Your XSQL Page

Use `<xsql:include-xml>` with a query to retrieve the CLOB value.

Combining JavaServer Pages and XSQL in the Same Page

Is it possible to combine XSQL and JavaServer Pages (JSP) tags in the same page or do you use include tags for that?

JSP and XSQL are two different models. JSP is a model that is based on writing streams of characters to an output stream. XSQL is a model that is pure XML and XSLT-based. At the end of the day, some result like HTML or XML comes back to the user, and there really isn't anything that you can implement with XSQL that you cannot implement in JSP by writing code and working with XML documents as streams of characters, doing lots of internal reparsing. XSQL fits the architecture when customers want to cleanly separate the data content (represented in XML) from the data presentation (represented by XSLT stylesheets). Since it specializes in this XML/XSLT architecture, it is optimized for doing that.

You can, for example, use `<jsp:include>` or `<jsp:forward>` to have a JSP page include/forward to an XSQL page. This is the best approach.

Choosing a Stylesheet Based on Input Arguments

It is possible to change stylesheets dynamically based on input arguments.

You can achieve this by using a lexical parameter in the href attribute of your xml-stylesheet processing instruction.

```
<?xml-stylesheet type="text/xsl" href="{@filename}.xsl"?>
```

The value of the parameter can be passed in as part of the request, or by using the `<xsql:set-page-param>` you can set the value of the parameter based on a SQL query.

Sorting the Result Within the Page

The following question was presented:

I have a set of 100 records, and I am showing 10 at a time. On each column name I have made a link. When that link is clicked, I want to sort the data in the page alone, based on that column.

If you are writing for IE5 alone and receiving XML data, you can use Microsoft's XSL to sort data in a page. If you are writing for another browser and the browser is getting the data as HTML, then you have to have a sort parameter in XSQL script and use it in `ORDER BY` clause. Just pass it along with the skip-rows parameter.

9

Pipeline Definition Language for Java

This chapter contains these topics:

Using Pipeline Definition Language

XML Pipeline definition Language from W3C, enables you to describe the processing relations between XML resources. A pipeline document specifies input and output of processes. A pipeline controller uses the pipeline document to execute the specified processes.

Oracle XML Pipeline Processor is built upon the XML Pipeline Definition Language Version 1.0, W3C Note 28 February 2002. The processor can take an input XML pipeline document and execute the pipeline processes according to the derived dependencies. The pipeline document is an XML document, and specifies the processes to be executed in a declarative manner. In addition to the XML Pipeline Processor, the XDK defines several Pipeline Processes which can be piped together in a pipeline document.

There are some differences between the W3C Note and the Oracle implementation. They are:

- The parser processes (`DOMParserProcess` and `SAXParserProcess`) are included in the XML pipeline (Section 1 of the note).
- Currently XML Base is not supported (Section 2.1)
- Only the final target output is checked to see if it is up-to-date with respect to the available pipeline inputs. The intermediate output of every process is not checked for being up-to-date. (Section 2.2).
- For the select attribute, anything between double-quotes "" is considered to be a string literal.
- The processor throws an error if more that one process produces the same infoset (Section 2.4.2.3).
- The document element is not supported, because it is redundant functionality (Section 2.4.2.8).

The Pipeline Definition Language is described at:

See Also: `http://www.w3.org/TR/xml-pipeline/`

Example of a Pipeline Definition Language Application

The files for this example are in `/xdk/demo/java/pipeline/`. The application `PipelineSample.java` calls the pipeline document (an instance of the Pipeline Definition Language) named `pipedoc.xml`, which names `book.xsl` as the stylesheet to be used, `myresult.html` as the output HTML file, and `book.xml` as the XML document to be parsed. The processes are p2, p3, and p1 in the order given in `pipedoc.xml`. However, the processes are run in the order p1, p2, p3.

To run this example:

1. Add `xmlparserv2.jar` and the current directory to the `CLASSPATH`.
2. Use `make` to generate `.class` files.
3. Run the sample program:

   ```
   make demo
   ```

   Or run:

   ```
   java PipelineSample pipedoc.xml pipelog seq
   ```

   The first argument (`pipedoc.xml`) is the required pipeline document. `pipelog` is an optional log file that you name. If omitted, the default log file created is `pipeline.log`.

   `seq` can be either "para" or "seq", entered without the quotes. "seq" requests sequential processing. If omitted, the default mode is processing parallel threads, the same mode as if you entered "para".
4. View the pipeline target created, in this case `myresult.html`.

The error handler called by the Java program is `PipelineSampleErrHdlr.java`.

Here is `book.xml`, the input XML document with which you started the processing:

```
<?xml version="1.0"?>
<booklist>
  <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>MaMillan Publishers</publisher>
    <price>9.99</price>
  </book>
</booklist>
```

Here is `pipedoc.xml`, the pipeline document:

```
<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"/>
     <input name="document" label="xmldoc"/>
     <output name="result" label="myresult.html"/>
   </process>

  <process id="p1" type="domparser.p" ignore-errors="true">
     <input name="xmlsource" label="book.xml "/>
     <output name="dom" label="xmldoc"/>
     <param name="preserveWhitespace" select="true"></param>
     <error name="dom">
       <html xmlns="http://www/w3/org/1999/xhtml">
         <head>
            <title>DOMParser Failure!</title>
         </head>
         <body>
           <h1>Error parsing document</h1>
         </body>
       </html>
     </error>
  </process>


</pipeline>
```

The stylesheet `book.xsl` is listed next:

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:output method="xml"/>

  <xsl:template match="/">
```

```
    <HTML>
      <HEAD>
      </HEAD>
      <xsl:apply-templates/>
    </HTML>
  </xsl:template>

  <!-- document xsl:template -->

  <xsl:template match="booklist">
      <BODY BGCOLOR="#CCFFFF">
        <H1>List of books</H1>
        <P> This will illustrate the transformation of an XML file containing
            a list of books to an HTML table form </P>
        <xsl:apply-templates/>
      </BODY>
  </xsl:template>

  <xsl:template match="booklist/book">
<BR><B><xsl:apply-templates/></B></BR>
  </xsl:template>

  <xsl:template match="booklist/book/title">
      <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="booklist/book/author">
     <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="booklist/book/publisher">
  </xsl:template>

  <xsl:template match="booklist/book/price">
       Price: $<xsl:apply-templates/>
  </xsl:template>


</xsl:stylesheet>
```

The output is `myresult.html` in the `/log` subdirectory

```
<?xml version = '1.0'?>
<HTML><HEAD/><BODY BGCOLOR="#CCFFFF"><H1>List of books</H1><P> This will
 illustrate the transformation of an XML file containing list of books to an
 HTML table form </P>
  <BR/>
    Twelve Red Herrings
    Jeffrey Archer


       Price: $7.99

  <BR/>
    The Eleventh Commandment
    Jeffrey Archer


       Price: $3.99

  <BR/>
```

```
    C++ Primer
    Lippmann


       Price: $4.99

  <BR/>
    Emperor's New Mind
    Roger Penrose


       Price: $15.9

  <BR/>
    Evening News
    Arthur Hailey


       Price: $9.99

</BODY></HTML>
```

The Command-line Pipeline Tool orapipe

The command-line pipeline tool is named `orapipe`. Before running it for the first time, add `xmlparserv2.jar` to your CLASSPATH. `orapipe` must have at least one argument, the pipeline document (`pipedoc.xml` in the code example presented in the preceding section).

To run orapipe, use the following syntax, where `pipedoc` is the required pipeline document you prepare in advance:

```
orapipe options pipedoc
```

Table 9–1 describes the available options:

Table 9–1 orapipe: Command-line Options

Option	Purpose
`-help`	Prints the help message
`-log` *`logfile`*	Writes errors and messages to the log file you name. The default is `pipeline.log`.
`-noinfo`	Do not log informational items. The default is on.
`-nowarning`	Do not log any warnings. The default is on.
`-validate`	Validate the input `pipedoc` with the pipeline schema. The default is do not validate.
`-version`	Prints the release version.
`-sequential`	Executes the pipeline in sequential mode. The default is parallel.

10
XDK JavaBeans

This chapter contains these topics:

Accessing Oracle XDK JavaBeans

The Oracle XDK JavaBeans are provided as part of XDK with the Enterprise and Standard Editions.

The following new JavaBeans were added in release 10.1:

- `XSDValidator` - encapsulates `oracle.xml.parser.schema.XSDValidator` class and adds capabilities for validating a DOM tree.
- `XMLCompress` - encapsulates XML compression functionality.
- `XMLDBAccess` - extension of `DBAccess` JavaBean to support the `XMLType` column in which XML documents are stored in an Oracle database table.

XDK JavaBeans facilitate the addition of graphical interfaces to your XML applications.

Bean encapsulation includes documentation and descriptors that can be accessed directly from Java Integrated Development Environments like JDeveloper.

> **See Also:** *Oracle XML API Reference* contains listings of the methods in all the JavaBeans.

Database Connectivity

Database Connectivity is included with the XDK JavaBeans. The beans can now connect directly to a JDBC-enabled database to retrieve and store XML and XSL files.

XDK JavaBeans Overview

XDK JavaBeans comprises the following beans:

DOMBuilder

The `DOMBuilder` JavaBean is a non-visual bean. It builds a DOM Tree from an XML document.

The `DOMBuilder` JavaBean encapsulates the XML Parser for Java's `DOMParser` class with a bean interface and extends its functionality to permit asynchronous parsing. By registering a listener, Java applications can parse large or successive documents and then allow control to return immediately to the caller.

See Also: "DOMBuilder JavaBean" on page 10-3

XSLTransformer

The `XSLTransformer` JavaBean is a non-visual bean. It accepts an XML file, applies the transformation specified by an input XSL stylesheet and creates the resulting output file.

`XSLTransformer` JavaBean enables you to transform an XML document to almost any text-based format including XML, HTML, and DDL, by applying the appropriate XSL stylesheet.

- When integrated with other beans, XSLTransformer JavaBean enables an application or user to view the results of transformations immediately.
- 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.

See Also: "XSLTransformer JavaBean" on page 10-4

DBAccess

DBAccess JavaBean maintains `CLOB` tables that contain multiple XML and text documents.

XMLDiff

The `XMLDiff` JavaBean performs a tree comparison on two XML DOM trees. It displays the two XML DOM trees and shows the differences between the XML trees. A node can be inserted, deleted, moved, or modified. Each of these operations is shown in a different color or style.

XMLCompress

This JavaBean is an encapsulation of the XML compression functionality. The supported functions are compression of the internal DOM tree obtained by means of a DOMParser, compression of the SAX events thrown by the SAX parser, and un-compression of the serialized XML data, returning an `XMLDocument` object.

XMLDBAccess

This JavaBean is an extension of the `DBAcess` bean to support the XMLType column, in which XML documents are stored in an Oracle database table. Methods are provided to list, delete, or retrieve `XMLType` instances and their tables.

XSDValidator

This JavaBean is a class file that encapsulates the `oracle.xml.parser.schema.XSDValidator` class and adds capabilities for validating a DOM tree.

DOMBuilder JavaBean

`DOMBuilder` class implements an XML 1.0 parser according to the World Wide Web Consortium (W3C) recommendation. It parses an XML document and builds a DOM tree. The parsing is done in a separate thread and the `DOMBuilderListener` interface must be used for notification when the tree is built.

Use for Asynchronous Parsing in the Background

The `DOMBuilder` bean encapsulates the XML Parser for Java with a bean interface. It extends its functionality to permit asynchronous parsing. By registering a listener, a Java application can parse documents and return control return to the caller.

Asynchronous parsing in a background thread can be used interactively in visual applications. For example, when parsing a large file with the normal parser, the user interface freezes until the parsing has completed. This can be avoided with the `DOMBuilder` bean. After calling the `DOMBuilder` bean parse method, the application can immediately regain control and display "Parsing, please wait". If a "Cancel" button is included you can also cancel the operation. The application can continue when domBuilderOver() method is called by `DOMBuilder` bean when background parsing task has completed.

When parsing a large number of files, `DOMBuilder` JavaBean can save time. Response times that are up to 40% faster have been recorded when compared to parsing the files one by one.

DOMBuilder JavaBean Usage

Figure 10–1 illustrates `DOMBuilder` JavaBean usage.

1. The XML document to be parsed is input as a file, string buffer, or URL.
2. This inputs the method `DOMBuilder.addDOMBuilderListener(DOMBuilderListener)` and adds DOMBuilderListener.
3. The `DOMBuilder.parser()` method parses the XML document.
4. Optionally, the parsed result undergoes further processing.
5. `DOMBuilder`Listener API is called using `DOMBuilderOver()` method. This is called when it receives an asynchronous call from an application. This interface must be implemented to receive notifications about events during asynchronous parsing. The class implementing this interface must be added to the `DOMBuilder` using `addDOMBuilderListener` method.

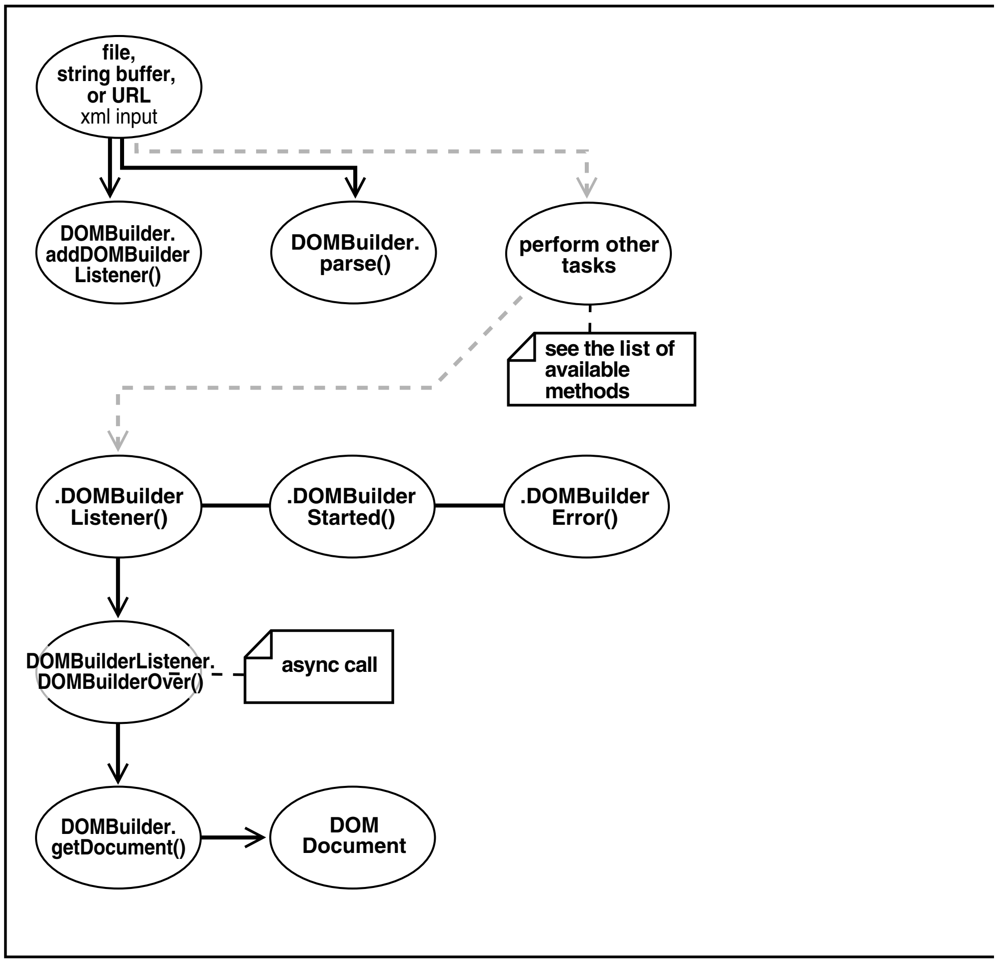
   Available `DOMBuilderListener` methods are:

   - `domBuilderError(DOMBuilderEvent)`. This method is called when parse errors occur.
   - `domBuilderOver(DOMBuilderEvent)`. This method is called when the parse completes.

- `domBuilderStarted(DOMBuilderEvent)`. This method is called when parsing begins.

6. `DOMBuilder.getDocument()` fetches the resulting DOM document and outputs the DOM document.

Figure 10–1 DOMBuilder JavaBean Usage


file, string buffer, or URL xml input
DOMBuilder. addDOMBuilder Listener()
DOMBuilder. parse()
perform other tasks
see the list of available methods
.DOMBuilder Listener()
.DOMBuilder Started()
.DOMBuilder Error()
DOMBuilderListener. DOMBuilderOver()
async call
DOMBuilder. getDocument()
DOM Document

XSLTransformer JavaBean

The `XSLTransformer` JavaBean accepts an XML file and applies the transformation specified by an input XSL stylesheet to create and output file. It enables you to transform an XML document to almost any text-based format, including XML, HTML, and DDL, by applying an XSL stylesheet.

When integrated with other beans, `XSLTransformer` JavaBean enables an application or user to immediately view the results of transformations.

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 `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.

XSL transformations can be time consuming. Use `XSLTransformer` bean in applications that transform large numbers of files and it can concurrently transform multiple files.

`XSLTransformer` bean can be used for visual applications for a responsive user interface. There are similar issues here as with `DOMBuilder`.

By implementing `XSLTransformerListener()` method, the caller application can be notified when the transformation is complete. The application is free to perform other tasks in between requesting and receiving the transformation.

XSLTransformer JavaBean: Regenerating HTML Only When Data Changes

This scenario illustrates one way of applying `XSLTransformer` JavaBean.

1. Create a SQL query. Store the selected XML data in a CLOB table.
2. Using the `XSLTransfomer` JavaBean, create an XSL stylesheet and interactively apply this to the XML data until you are satisfied with the data presentation. The output can be HTML produced by the XSL transformation.
3. Now that you have the desired SQL (data selection) and XSL (data presentation), create a trigger on the table or view used by your SQL query. The trigger can execute a stored procedure. The stored procedure, do the following:
   - Run the query
   - Apply the stylesheet
   - Store the resulting HTML in a `CLOB` table
4. This process can repeat whenever the source data table is updated.

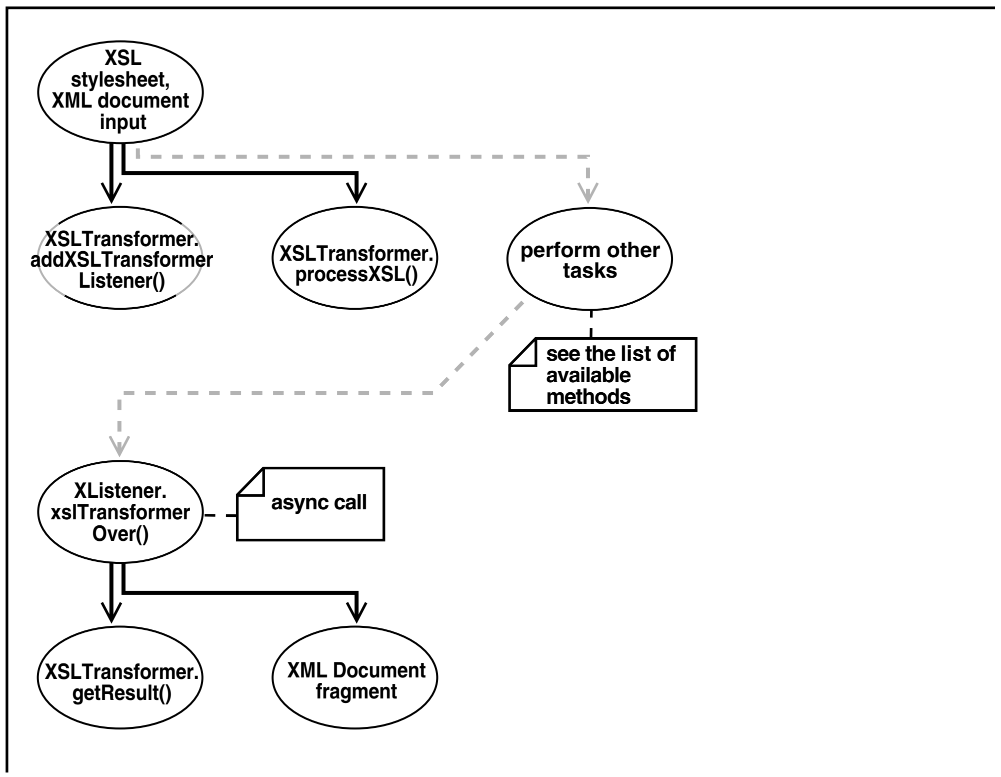
   The HTML stored in the `CLOB` table always mirrors the last data stored in the tables being queried. A JSP (JavaServer Page) can display the HTML.

   In this scenario, multiple end users do not produce multiple data queries that contribute to increased use of the database. The HTML is regenerated only when the underlying data changes.

How to Use XSLTransformer JavaBean

Figure 10–2 illustrates `XSLTransformer` bean usage.

Figure 10–2 XSLTransformer JavaBean Usage


XSL stylesheet, XML document input
XSLTransformer. addXSLTransformer Listener()
XSLTransformer. processXSL()
perform other tasks
see the list of available methods
XListener. xslTransformer Over()
async call
XSLTransformer. getResult()
XML Document fragment

1. An XSL stylesheet and XML document are input to the `XSLTransformer` using the `XSLTransfomer.addXSLTransformerListener (XSLTransformerListener)method`. This adds a listener.
2. The `XSLTransfomer.processXSL()` method initiates the XSL transformation in the background.
3. Optionally, other work can be assigned to the `XSLTransformer` bean.
4. When the transformation is complete, an asynchronous call is made and the `XSLTransformerListener.xslTransformerOver()` method is called. This interface must be implemented to receive notifications about events during the asynchronous transformation. The class implementing this interface must be added to the `XSLTransformer` event queue using the method `addXSLTransformerListener`.
5. The `XSLTransformer.getResult()` method returns the XML document fragment for the resulting document.
6. It outputs the XML document fragment.

DBAccess JavaBean

`DBAccess` JavaBean maintains `CLOB` tables that can hold multiple XML and text documents. Each table is created using the following statement:

```
CREATE TABLE tablename FILENAME CHAR(16) UNIQUE, FILEDATA CLOB) LOB(FILEDATA)
   STORE AS (DISABLE STORAGE IN ROW)
```

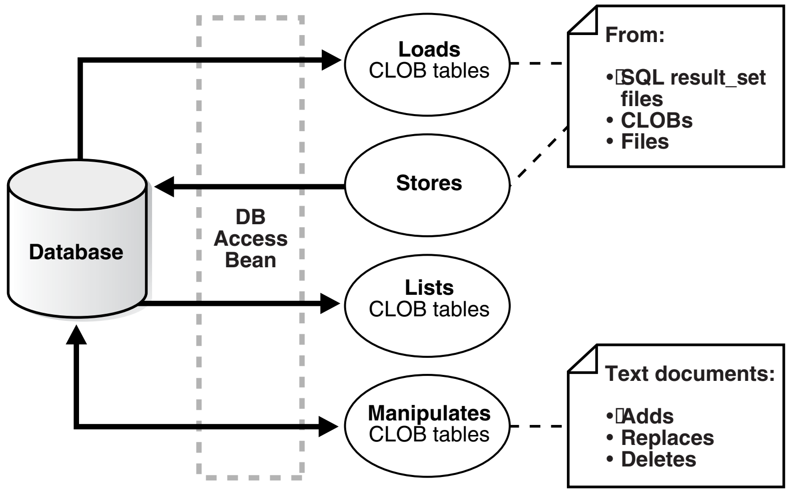
Each XML (or text) document is stored as a row in the table. The FILENAME field holds a unique string used as a key to retrieve, update, or delete the row. Document text is stored in the FILEDATA field. This is a `CLOB` object. `DBAccess` bean does the following tasks:

- Creates and deletes `CLOB` tables
- Lists a `CLOB` table's contents
- Adds, replaces, or deletes text documents in the `CLOB` tables

DBAcess JavaBean Usage

Figure 10–3 illustrates the `DBAccess` bean usage. It shows how `DBAccess` bean maintains, and manipulates XML documents stored in `CLOB`s.

Figure 10–3 DBAccess JavaBean Usage Diagram



XMLDiff JavaBean

The `XMLDiff` JavaBean performs a tree comparison on two XML DOM trees. It displays the two XML trees and shows the differences between the XML trees. A node can be inserted, deleted, moved, or modified. Each of these operations is shown in a different color or style as in the following list:

- Red—Used to show a modified Node or Attribute
- Blue—Used to show a new Node or Attribute
- Black—Used to show a deleted Node or Attribute

Moves will be displayed visually as a delete or insert operation.

You can generate the differences between the two XML trees in the form of XSL code. The first XML file can be transformed into the second XML file by using the XSL code generated.

Note: Currently you cannot customize the GUI display.

XMLCompress JavaBean

This bean class is a simple encapsulation of the XML Compression functionality. The capabilities that are supported in this class are essentially compression of the internal DOM tree obtained via a DOMParser,

Compression of the SAX events thrown by the SAX Parser, decompression of the serialized XML data, returning an `XMLDocument` object. The input for compression can be from an InputStream, a Java string, a database `CLOB` object, or an `XMLType` object. In all cases the `outputStream` has to be set beforehand so that the compressed data is written to it. If the input data is unparsed, the parsing for it is done with no validation.

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 done using the `java.io.File` objects as parameters.

XMLDBAccess JavaBean

This bean is an extension of the `DBAcess` bean to support the `XMLType` column, in which XML documents are stored in an Oracle database table. Methods are provided to list, delete, or retrieve `XMLType` instances and their tables.

XSDValidator JavaBean

This class file encapsulates the `oracle.xml.parser.schema.XSDValidator` class and adds capabilities for validating a DOM tree. The schema document is a constant and the validation is done for XML Documents that can be passed as InputStreams, URLs, and so on.

The validation is done only after the DOM tree is built in all the cases. Nodes with errors are returned in a vector of stack trees where the top element of the stack represents the root node and child nodes are obtained by popping the elements of the stack.

JavaBean Examples

The XDK JavaBean sample directory, `/xdk/demo/java/transviewer/`, contains sample applications that illustrate how to use JavaBeans.

Table 10–1 lists the example files. The subsequent sections show how to install these samples.

Table 10–1 JavaBean Example Files

File Name	Description
`AsyncTransformSample.java.`	Sample nonvisual application using `XSLTransformer` bean and `DOMBuilder` bean. It applies the XSLT stylesheet specified in `doc.xsl` on all `.xml` files from the current directory. The results are in the files with extension `.log`.
`XMLDBAccessSample.java`	A non-GUI sample for the `XMLDBAccess` bean which demonstrates the way that the `XMLDBAccess` bean APIs can be used to store and retrieve the XML documents inside the database, using XMLType tables. To use `XMLType`, an Oracle9*i*, or later, installation is necessary along with the `xdb.jar`. `Sample5` will run `XMLDBAccessSample` using the values for `HOSTNAME`, `PORT`, `SID`, `USERID`, and `PASSWORD` as defined in the `Makefile`. These must be modified if required. The file `booklist.xml` is used to insert data into the database. The output is copied to `xmldbaccess.log`.
`XMLDiffSample.java`	Sample visual application that uses `XMLDiff` bean to find differences between two XML files and generate an XSL stylesheet. This stylesheet can be used to transform the first input XML into the second input XML file. See "XMLDiffSample.java" on page 10-10.
`compviewer.java`	Sample visual application that uses `XMLCompress` bean to compress an XML file or XML data from the database obtained through SQL query or from a CLOB or an XMLType Table. The application also lets you decompress the compressed stream and view the resulting DOM tree.
`XSDValidatorSample.java`	Sample application for `XSDValidator` bean. It takes two arguments as input, an XML file and a schema file. The error occurring during validation, including line numbers, are displayed. See "XSDValidatorSample.java" on page 10-10.

Installing the JavaBean Examples

The JavaBeans require, as a minimum, JDK 1.2.2.

Here are the steps you take to generate the sample executables:

1. Download and install the following components used by the XDK JavaBeans:
   - Oracle JDBC Driver for thin client (file `classes12.zip`)
   - Oracle XML SQL Utility `(xsu12.jar)`
   - `JAR` file containing the `XMLType` definitions `(file xdb.jar)`

   After installing these components, include the files in your `CLASSPATH`.
2. Change `JDKPATH` in `Makefile` to point to your JDK path. If you do not have an `ORACLE_HOME` set, then set it to the root directory of your XDK JavaBeans installation.
3. Generate `.class` files. Run the sample programs using the following commands, which use labels in the file `Makefile`:

```
gmake sample3
gmake sample5
gmake sample6
gmake sample7
gmake sample10
```

See Also: README contains details of the various programs labelled sample3 through sample10.

XMLDiffSample.java

`Sample6` is a demo for `XMLDiff` JavaBean. It invokes a GUI which allows you to choose the input data files from the `File` menu using '`Compare XML Files`' item. The XSLT generated can be applied on input XML `file1` using `Transform` menu. The resulting XML file (which is the same as input `file2`) can be saved using '`Save`' `As` item under `File` menu. By default, the two XML files `XMLDiffData1.txt` and `XMLDiffData2.txt` are compared and the output XSLT is stored as `XMLDiffSample.xsl`.

If the input XML files have a DTD which accesses a URL outside the firewall, then modify `XMLDiffSample.java` to include the proxy server settings before the `setFiles()` call:

```
/* 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 also have to import `java.util.*;`

XSDValidatorSample.java

`Sample10` is a demonstration for the `XSDValidator` JavaBean. It takes as default the data file `purchaseorder.xml` and the `purchaseorder.xsd` schema file. The output displays the validation errors.

11 Using XDK and SOAP

This chapter contains these topics:

What Is SOAP?

The term *Web Services* is used to describe a functionality made available by an entity over the Web. It is an application that uses XML standards and is published, located and executed through the Web.

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 simple transport protocols such as HTTP, it is not blocked by firewalls and is very easy to use. SOAP is independent of operating system, implementation language, and any single object model.

SOAP supports remote procedure calls. Its messages are only of the three types:

- a request for a service, including input parameters
- a response to the requested service, including return value and output parameters
- an optional fault element containing error codes and information

SOAP messages consist of:

- an *envelope* that contains the message, defines how to process the message, who should process the message, and whether processing is optional or mandatory. This is a required part.
- *encoding rules* that describe the datatypes for the application. These rules define a serialization mechanism that converts the application datatypes to XML and XML to datatypes.
- *remote procedure* call and responses definitions. This is called a body element and is a required part.

SOAP 1.1 specification is a World Wide Web Consortium (W3C) note. The W3C XML Protocol Working Group has been formed to create a standard that will supersede SOAP 1.1. Oracle is a member of this group. The standard will be called SOAP 1.2.

A SOAP service remote procedure call (RPC) request and response sequence includes the steps:

1. A SOAP client writes a request for service in a conforming XML document, using either an editor or the Oracle SOAP client API.
2. The client sends the document to a SOAP Request Handler running as a servlet on a Web server.
3. The Web Server dispatches the message as a service request to an appropriate server-side application providing the requested service.
4. The application must verify that the message contains supported parts. The response from the service is returned to the SOAP Request Handler servlet and then to the caller using the SOAP payload format.

See Also:

- `http://www.w3.org/TR/SOAP/`
- `http://xml.apache.org/soap`

What Are UDDI and WSDL?

The Universal Description, Discovery and Integration (**UDDI**) specification provides a platform-independent framework using XML to describe services, discover businesses, and integrate business services on the Internet. The UDDI business registry is the public database where companies are registered. The UDDI business registration is an XML file with three sections:

- *white pages* that include address, contact, and known identifiers
- *yellow pages* include industrial categorization
- *green pages* containing the technical information about exposed services

The Web Services Description Language (**WSDL**) is a general purpose XML language for describing the interface, protocol bindings, and deployment details of Web Services. WSDL provides a method of describing the abstract interface and arbitrary network services. A WSDL service is registered or embedded in the UDDI registry.

The stack of protocol stack used in Web Services is summarized in the following list:

- Universal Service Interoperability Protocols (WSDL, and so on.)
- Universal Description, Discovery Integration (UDDI)
- Simple Object Access Protocol (SOAP)
- XML, XML Schema
- Internet Protocols (HTTP, **HTTPS**, TCP/IP)

What Is Oracle SOAP?

Oracle SOAP is an implementation of the Simple Object Access Protocol. Oracle SOAP is based on the SOAP open source implementation developed by the Apache Software Foundation.

How Does Oracle SOAP Work?

Consider this example: a `GetLastTradePrice` SOAP request is sent to a `StockQuote` service. The request takes a string parameter, the company stock symbol, and returns a float in the SOAP response. The XML document represents the SOAP message. The SOAP envelope element is the top element of the XML document. XML namespaces are used to clarify SOAP identifiers from application-specific identifiers. The following example uses HTTP as the transport protocol. The rules governing XML payload format in SOAP are independent of the fact that the payload is carried in HTTP. The SOAP request message embedded in the HTTP request is:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
<SOAP-ENV:Envelope  xmlns:SOAP-   ENV="http://schemas.xmlsoap.org/soap/
envelope/"  SOAP-
ENV:encodingStyle="http://schemas.xnlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<m:GetLastTradePrice xmlns:m="Some-URI">
<symbol>ORCL</symbol>
<m:GetLastTradePrice>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Here is the response HTTP message:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope  xmlns:SOAP-
ENV=http://schemas.xmlsoap.org/soap//envelope/ SOAP-
ENV:encodingStyle="http://schemas.xnlsoap.org/soap/encoding/"/>
<SOAP-ENV:Body>
<m:GetLastTradePriceResponse xmlns:m="Some-URI">
<Price>34.5</Price>
</m:GetLastTradePriceResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Oracle SOAP and IDAP

IDAP is an XML-based specification to perform AQ operations. SOAP defines a generic mechanism to invoke a service. IDAP defines these mechanisms to perform AQ operations.

IDAP has the following key properties not defined by SOAP:

- Transactional behavior - You can perform AQ operations in a transactional manner. Your transaction can span multiple IDAP requests.
- Security - All the IDAP operations can be done only by authorized and authenticated users.
- You can also perform AQ operations through the SOAP interface. AQ encapsulates operations in IDAP format.
- Character set transformations - This is a very important requirement for any communication. Internet user's machine may have different character set id than the server machine.
- Extensible AQ Servlet for AQ Internet operations - The AQ servlet performing AQ operations is extensible. You can specify time-out, connection pooling, TAF, apply XML stylesheets, perform post AQ and pre-AQ database operations in the AQ Servlet.

There is no difference in SOAP and IDAP access to AQ except the line specifying the namespace for the envelope.

For IDAP that line is:

```
<Envelope xmlns="http://ns.oracle.com/AQ/schemas/envelope">
```

While in SOAP, it is:

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
```

Everything else remains the same.

What Is a SOAP Client?

A SOAP client application represents a user-written application that makes SOAP requests. The SOAP client has these capabilities:

- Gathers all parameters that are needed to invoke a service.
- Creates a SOAP service request message. This is an XML message that is built according to the SOAP protocol and that contains all the values of all input parameters encoded in XML. This process is called *serialization*.
- Submits the request to a SOAP server using some transport protocol that is supported by the SOAP server.
- Receives a SOAP response message.
- Determines the success or failure of the request by handling the SOAP Fault element.
- Converts the returned parameter from XML to native datatype. This process is called *deserialization*.
- Uses the result as needed.

SOAP Client API

SOAP clients generate the XML documents that compose a request for a SOAP service and handle the SOAP response. Oracle SOAP processes requests from any client that sends a valid SOAP request. To facilitate client development, Oracle SOAP includes a SOAP client API that provides a generic way to invoke a SOAP service.

The SOAP client API supports a synchronous invocation model for requests and responses. The SOAP client API makes it easier for you to write a Java client application to make a SOAP request. The SOAP client API encapsulates the creation of the SOAP request and the details of sending the request over the underlying transport protocol. The SOAP client API also supports a pluggable transport, allowing the client to easily change the transport (available transports include HTTP and HTTPS).

What Is a SOAP Server?

A SOAP server has the following capabilities:

- The server receives the service request.
- The server parses the XML request and then decides to execute the message or reject it.
- If the message is executed, the server determines if the requested service exists.

- The server converts all input parameters from XML into datatypes that the service understands.
- The server invokes the service.
- The return parameter is converted to XML and a SOAP response message is generated.
- The response message is sent back to the caller.

Oracle SOAP Security Features

Oracle SOAP uses the security capabilities in the transport to support secure access and to support other security features. For example, using HTTPS, Oracle SOAP provides confidentiality, authentication, and integrity over the Secure Sockets Layer (SSL). Other security features such as logging and authorization are provided by the service provider.

SOAP Transports

SOAP transports are the protocols that carry SOAP messages. Oracle SOAP supports the following transports:

- HTTP: This protocol is the basic SOAP transport. The Oracle SOAP Request Handler Servlet manages HTTP requests and supplies responses directly over HTTP. This protocol is becoming a standard because of its popularity.
- HTTPS: The Oracle SOAP Request Handler Servlet manages HTTPS requests and supplies responses, with different security levels supported.

Administrative Clients

SOAP administrative clients include the Service Manager and the Provider Manager. These administrative clients are services that support dynamic deployment of new services and new providers.

SOAP Request Handler

The SOAP Request Handler is a Java servlet that receives SOAP requests, looks up the appropriate service provider, handles the service provider that invokes the requested method (service), and returns the SOAP response, if any.

SOAP Provider Interface and Providers

Oracle SOAP includes a provider implementation for Java classes. Other providers can be added.

Provider Interface

The provider interface allows the SOAP server to uniformly invoke service methods regardless of the type of provider (Java class, stored procedure, or some other provider type). There is one provider interface implementation for each type of service provider, and it encapsulates all provider-specific information. The provider interface makes SOAP implementation easily extensible to support new types of service providers.

Provider Deployment Administration

Oracle SOAP provides the provider deployment administration client to manage provider deployment information.

SOAP Services Provided

SOAP application developers provide SOAP services. These services are made available using the supplied default Java class provider or custom providers. Oracle SOAP includes a service deployment administration client that runs as a service to manage SOAP services. SOAP services, including Java services, represent user-written applications that are provided to remote SOAP clients.

Advantages of XML Over EDI

Here are facts about Electronic Data Interchange (EDI):

- EDI is a difficult technology: EDI enables machine-to-machine communication in a format that developers cannot easily read and understand.
- EDI messages are very difficult to debug. XML documents are readable and easier to edit.
- EDI is not flexible: It is very difficult to add a new trading partner as part of an existing system; each new trading partner requires its own mapping. XML is extremely flexible and has the ability to add new tags on demand and to transform an XML document into another XML document, for example, to map two different formats of purchase order numbers.
- EDI is expensive: developer training costs are high, and deployment of EDI requires very powerful servers that need a specialized network. EDI runs on Value Added Networks, which are expensive. XML works with inexpensive Web servers over existing Internet connections.

See Also: For more information about Oracle SOAP and Web Services, including documentation and downloads, see:

- `http://www.oracle.com/technology/documentation/ias.html` for the OracleAS SOAP Developer's Guide
- *Oracle Simple Object Access Protocol Developer's Guide*
- *Oracle Streams Advanced Queuing User's Guide and Reference* for a discussion of Internet access to AQ
- *Oracle XML API Reference*
- The SOAP API is on the Product CD, Disk 1, in file `doc/readmes/ADDEN_rdbms.htm`

SOAP Example

Consider an enterprise or government entity that has inventories to be maintained at its headquarters, and at multiple remote branches. The SOAP solution to be described in this chapter considers the headquarters as a message server and the branches as message clients. Among the several tasks to be performed are:

- Branch registration. Each branch must be known to headquarters for inventory updates to be made.
- Branch inventory management based on sales. The branches order more supplies from headquarters when the item in the branch inventory is low.
- Headquarters (HQ) inventory monitoring. The branches must be informed of new items that headquarters has added to its inventory stock.

Consider HQ inventory monitoring and the task of informing the branches when a new item is added to the HQ inventory. This example solves this problem with SOAP messaging and several XML features.

The branches can be using non-Oracle databases, because SOAP is not dependent on any DBMS. You can generalize this example so that it can be used for communicating with customers, suppliers, or other entities that are not part of your organization.

XML Features Used in the SOAP Example

The SOAP messages employ the following features:

- Advanced Queuing (AQ). This Oracle feature uses asynchronous communications between applications and users, with transaction control and security. AQ keeps users from being blocked when they enter new inventory items at HQ.
- AQ provides the Java Messaging Service (JMS) APIs to enqueue and dequeue the messages.
- Columns with datatype `XMLType` are used to store SOAP messages in database tables, so that data about new items will not be lost.
- XML Compression reduces the payload size and speeds the transmission of messages.
- XSQL Servlet is used to publish content and for interacting with administrators.
- The message server at HQ invokes remote procedure calling (RPC).
- The SOAP call generates an HTTP request, encapsulates the inventory update request in a SOAP message, and invokes the SOAP service on all branches.
- Each branch either returns a confirmation or returns a fault (defined in the SOAP standard) to the message server.

Prerequisite Software for the SOAP Example

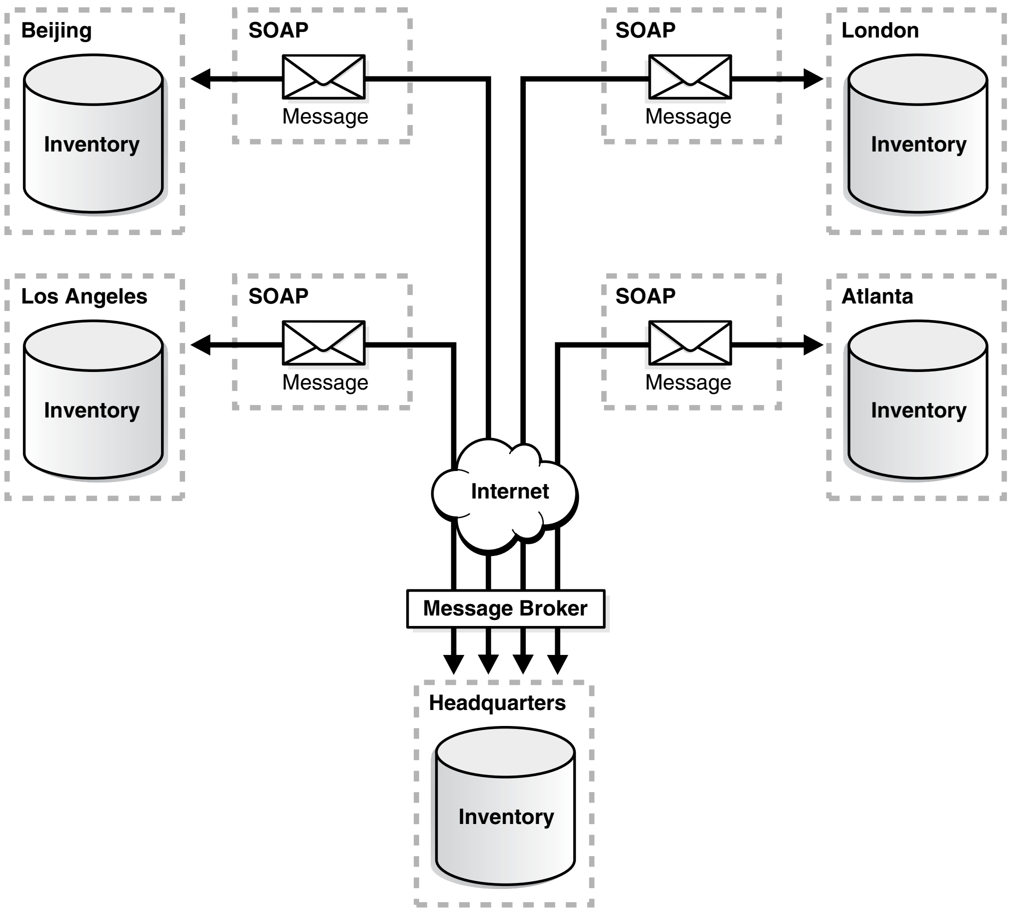
- Oracle Database
- XML Developer's Kit (XDK), Java components
- OC4J
- The SOAP example, "Build an XML-Powered Distributed Application", can be downloaded from the OTN:

  See Also: Download the SOAP example at `http://www.oracle.com/technology/tech/xml/`

How Is the SOAP Example Implemented?

An overview of the distributed inventory application is shown in the following illustration:

Figure 11–1 Using Soap in a Distributed Inventory Application



SOAP is used to manage inventory at each of the remote branches and at headquarters. The headquarters inventory application is a message server and the branch inventory application is a message client, each using SOAP services.

When headquarters adds a new item to its inventory, it broadcasts a message to all remote client branches that have registered with the message server. A message broker creates a message and pushes it onto the message queue, using AQ. AQ enables asynchronous communications between user applications, providing transaction control, security and preventing blocking of the entry of new items by the users at headquarters.

A message dispatcher process, which is listening to the message queue, detects the enqueued messages and calls the branches' SOAP services to dequeue the message and to update their local inventories.

The messages are stored in the database with complete logging information. The SOAP messages are stored in `XMLType` datatype instances and are thus a record of sent and received messages. This insures data integrity in the inventories.

At headquarters a table is created initially that has three columns: an identification number, a message of datatype `XMLType`, and a creation time.

XML Compression is another technology used to lower payloads, making throughput greater in large applications.

Setting Up the Tables and the SOAP Service

To store inventory data, log messages, and perform message queuing, run the `createdb.sql` script from the downloaded source files to set up database schemas for headquarters and a branch. This script calls a set of SQL scripts that creates a headquarters user and a branch user with proper privileges. It also creates tables and triggers for storing inventory information and messages and inserts sample data in both schemas.

In the headquarters user schema, create a table named `message_out_type`. This stores the inventory information update broadcast messages from headquarters to the branches. There are three columns in the table: `ID`, `MESSAGE`, and `CREATE_TIME`. The `MESSAGE` column datatype is `XMLType`.

Next, run the PL/SQL procedure, `CREATE_QUEUE`, that sets up message queues at both headquarters and the branches. This procedure uses functions in the `DBMS_AQADM` package to create the queue table, create the queue, and start the queue. Once the queue is started, enqueuing and dequeuing operations on the message queue are enabled.

The following PL/SQL procedure uses the `SYS.AQ$_JMS_BYTES_MESSAGE` message type to manage the compressed SOAP messages. This creates a queue called `broadcastb_queue` at the headquarters location:

```
begin
    create_queue('broadcastb_queue_tbl',
    'broadcastb_queue',
    'SYS.AQ$_JMS_BYTES_MESSAGE');
end;
```

The `inventoryBranchServer Java` class is the branch's service for inserting a new item into the branch inventory. When this service program receives a SOAP request, it decompresses the request content and saves it in the `inventory_item` table, using Oracle XML SQL Utility (XSU) to insert the item into the database. Oracle XSU creates canonical mappings between the XML document and database schema to perform SQL data operations. See the file

```
client/src/oracle/xml/pm/demo/branch/service/inventoryServer.java
```

in the downloaded software.

Requesting SOAP Service

The application makes requests to the headquarters SOAP service using a servlet called `insertItemServlet`, a Java class that extends `HttpServlet`. This servlet inserts a new item in the headquarters inventory.

The servlet request uses XSQL pages and the user's input in the application's Web interface (click "New Items") to generate an XML document. Oracle XSU then directly inserts the XML content into the database. The `insertItemServlet` performs several actions. For example, to broadcast an update message to the branches, it:

- Initializes the Message Dispatcher process.
- Compresses the XML document by calling the `CompressionAgent` class.
- Creates a SOAP message and stores it in the message logging table.
- Pushes the compressed XML document onto the message queue (enqueue).

Initializing the MessageDispatcher Process

When it is first called, `insertItemServlet` initializes the `MessageDispatcher` object. This object is stored in the `ServletContext` when the process is successfully initialized. This code initializes the `MessageDispatcher` object:

```
ServletContext context = getServletContext();
MessageDispatcher msgDispatcher =
    (MessageDispatcher)context.getAttribute("Dispatcher");

if (msgDispatcher == null) {
   System.out.println("Initialize Receiver.");
   msgDispatcher = new MesageDispatcher();
   context.setAttribute("Dispatcher",msgDispatcher);
   }
```

The `MessageDispatcher` Java class creates a `MessageBroker`, which in turn, creates a `MessageClient` to monitor each message queue and dispatch messages to the registered branches.

Compressing the XML Document

The following code from `insertItemServlet` creates the Compression Agent:

```
CompressionAgent cagent = new
   CompressionAgent("oracle:compression");
byte [] input = cagent.compression(m_content);
```

Creating a SOAP Message

The message is stored in a column defined as `XMLType`. The code from insertItemServlet that creates the SOAP message and stores it in the `MESSAGE_OUT_XMLTYPE` table is:

```
OraclePreparedStatement pstmt =
    (OraclePreparedStatement) conn.prepareStatement (
          "Insert into message_out_xmltype(message) values(?)");

m_content=createSOAPMessage(m_content);
oracle.xdb.XMLType xt = oracle.xdb.XMLType.createXML(conn,m_content);

pstmt.setObject(1, xt);
pstmt.execute();
```

Using `XMLType` lets us use XPATH with the `sys.XMLType.extract()` member function to query the portions of the message documents:

```
select e.message.extract('//item_info')
   .getStringVal() as result
from message_out_xmltype;
```

Enqueuing the XML Document

The following code from `insertItemServlet` creates the `MessageBroker` and enqueues the message:

```
MessageBroker mesgBroker =
   new MessageBroker("host_name",
   "Oracle_SID", "port_num",
   "thin", "cm", "cm", "broadcastb_queue");
mesgBroker.enqueueMessage(input);
```

When `insertItemServlet` finishes, the message is pushed onto the message queue and the Oracle AQ and `MessageDispatcher` processes update the branch inventory information. This ensures that the headquarters inventory system is not blocked during the branch system updates.

Listing of the Java Source File inserItemServlet.java

This file is found at `./server/src/insertItemServlet.java`:

```
/**
 * FileName: insertItemServlet.java
 * Description:
 *   Insert new Inventory Item into HQ database and broadcase the message to
 *   the branches.
 */
import java.io.*;
import java.util.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

// XSU
import org.w3c.dom.*;
import oracle.xml.parser.v2.*;
import oracle.xml.sql.dml.OracleXMLSave;

// XMLType
import oracle.xdb.XMLType.*;
import oracle.jdbc.driver.*;
import oracle.sql.*;

// SOAP Message
import oracle.AQ.*;
import oracle.xml.pm.queue.*;
import oracle.xml.pm.compression.CompressionAgent;

// Configuration
import oracle.xml.pm.util.ConfigManager;

/**
 * This class implements Message Borker
 */
public class insertItemServlet extends HttpServlet
{
  String m_content=null;
  String m_dblink = null;
  String m_usr = null;
  String m_passwd = null;
  String m_hostname = null;
  String m_sid = null;
  String m_port = null;

  /**
   * Initialize global variables
   * @param config - ServletConfig
   * @exeception - ServletException thrown if super.init fails
   */
  public void init(ServletConfig config) throws ServletException
  {
    super.init(config);
    // Initialize the JDBC Connection from Configuration Files
    try
    {
      ConfigManager xml_config = new ConfigManager("DEMOConfig.xml","cm");
      m_dblink = xml_config.dblink;
      m_usr= xml_config.usr;
      m_passwd = xml_config.passwd;
      m_hostname = xml_config.hostname;
      m_sid = xml_config.db_sid;
      m_port = xml_config.db_port;
    }
    catch(Exception ex)
    {
      // ex.printStackTrace();
      throw new ServletException(ex.getMessage());
    }
  }

  /**
   * HTTP Get
   * @param req - HttpServletRequest
   * @param res - HttpServletResponse
   * @exeception - IOException, ServletException
   */
   public void doGet(HttpServletRequest req, HttpServletResponse res)
       throws IOException, ServletException
  {
     doPost(req,res);
  }

  /**
   * HTTP POST
   * @param req - HttpServletRequest
   * @param res - HttpServletResponse
   * @exeception - IOException, ServletException
   */
  public void doPost(HttpServletRequest req, HttpServletResponse res)
  throws IOException, ServletException
  {
    ServletContext context = getServletContext();

    // Initialize MessageDispatcher for broadcast messages
    MessageDispatcher msgDispatcher =
      (MessageDispatcher) context.getAttribute("MessageDispatcher");

    if(msgDispatcher == null)
    {
        msgDispatcher = new MessageDispatcher("broadcastb_queue",m_hostname,
        m_sid, m_port, m_usr,m_passwd,m_dblink);
        context.setAttribute("MessageDispatcher",msgDispatcher);
     }

    // Initialize MessageBroker for broadcasting messages
    MessageBroker msgBroker = (MessageBroker)
                                  context.getAttribute("MessageBroker");
    if(msgBroker == null)
    {
      try
      {
        msgBroker = new MessageBroker(m_hostname, m_sid, m_port, "thin",m_usr,
                                      m_passwd,"broadcastb_queue",m_dblink);
        context.setAttribute("MessageBroker",msgBroker);
      }
      catch(Exception ex)
      {
        System.out.println("Error:"+ex.getMessage());
      }
    }

    PrintWriter out = res.getWriter();
    m_content = req.getParameter("content");

    // Save new Item information into database
    try
    {
      Connection conn = getConnection();

      OracleXMLSave sav = new OracleXMLSave(conn,"inventory_item_view");
      sav.insertXML(m_content);
      sav.close();
      conn.close();

      out.println("Insert Successful\n");
    }
    catch(Exception e)
    {
      out.println("Exception caught "+e.getMessage());
      return;
    }

    // Create and Enqueue the Message
    byte[] input = createMessage(m_content);
    msgBroker.enqueueBytesMessage(input);

    return;
  }

  // Subject to change to validate and using XML Update language
  // Since this message is not public we keep it with simplified SOAP format
  public byte[] createMessage(String content)
  {
     String message = null;

     message="<Envelope>"+
            "<Header>"+
            "<branch_sql>"+"select id,soapurl from branch"+"</branch_sql>"+
            "<objURI>"+"inventoryServer"+"</objURI>"+
            "<method>"+"addItem"+"</method>"+
            "</Header>"+
            "<Body>"+content+"</Body>"+
            "</Envelope>";

    // Compress the Message Content
    CompressionAgent cagent = new CompressionAgent("oracle:xml:compression");
    byte [] input = cagent.compress(message);

    return input;
  }
```

```
  /**
   * Get JDBC Connection
   * @return Connection - JDBC Connection
   * @exception SQLException - thrown if the connection can't be gotten.
   */
  public Connection getConnection() throws SQLException
  {
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
    Connection conn =DriverManager.getConnection (m_dblink,m_usr,m_passwd);
    return conn;
  }
}
```

Queuing Messages with AQ

The `MessageBroker` class is the agent that communicates with the message queue and invokes SOAP Remote Procedure Call (RPC).

The `MessageBroker` provides the following functionality:

Message Enqueuing and Dequeuing. AQ provides standard Java Message Service (JMS) APIs to enqueue and dequeue the messages. Before these operations, however, you need to get a `QueueConnection`, create a `QueueSession` with it, and get the message queue. During each `QueueSession`, `QueueSessioncommit()` is used for transactional control. If anything goes wrong when our messaging system sends messages to the branches, the commit will not occur and the message will remain in the database.

Invoking Remote SOAP Services. `MessageBroker` invokes the remote SOAP service with `RPCCall` Agent (in the `RPCCall` Java class). The SOAP `RPCCall` creates a Call object to specify the SOAP service and adds parameters to the parameters vector for the `Call` object. The `Call` object will invoke the remote SOAP service and return "Fault code" if there are errors. We define a Service object in the Service Java class to describe the Service information.

The SOAP call will generate the HTTP request, encapsulate the inventory update request in a SOAP message, and invoke the SOAP service on all branches. When headquarters calls a branch inventory service, the branch sends a confirmation back to it.

XML Compression

To make the messaging system more efficient, use XML compression. This lets you compress or decompress a message by using Java object serialization, such as:

```
XMLDocument.writeExternal(...);
XMLDocument.readExternal(...);
```

Our sample application uses the `CompressionAgent` Java class to provide the compress and decompress functionality. `CompressionAgent` includes both compress and decompress methods. The compress method returns a byte array and the decompress method returns an XML document.

Listing of the Java Source File MessageBroker.java

This lengthy source file is found at `./xmlagents/src/oracle/xml/pm/queue`

Some of the more important functions are `sendSOAPMessage()`, which sends SOAP files and `dequeueTextMessage()`, which dequeues information for each inventory item.

Summary of the SOAP Example

Using AQ and XDK in a SOAP messaging system can greatly improve reliability and performance. Inventory update messages are delivered quickly. If they cannot be delivered, they are stored and logged for sending again. These features provide reliable and asynchronous message queuing and speed the transfer of XML message data.

12 TransX Utility

This chapter contains these topics:

Overview of the TransX Utility

The TransX Utility simplifies the loading of translated seed data and messages into a database. It also reduces globalization costs by:

- Preparing strings to be translated.
- Translating the strings.
- Loading the strings to the database.

The TransX Utility minimizes translation data format errors and accurately loads the translation contents into pre-determined locations in the database. Other advantages of the TransX Utility are:

- Translation vendors no longer have to work with unfamiliar SQL and PL/SQL scripts.
- Syntax errors due to varying Globalization Support settings are eliminated.
- The UNISTR construct is no longer required for every piece of `NCHAR` data.

Development groups that need to load translated messages and seed data can use the TransX Utility to simplify what it takes to meet globalization requirements. Once the data is in a predefined format, the TransX Utility validates its format.

Choosing the correct encoding when loading translated data is automated because loading with TransX takes advantage of XML which describes the encoding. This means that loading errors due to incorrect encoding is impossible as long as the data file conforms to the XML standard.

Primary TransX Utility Features

This section describes the following features of the TransX Utility:

- Simplified Multilingual Data Loading
- Simplified Data Format Support and Interface

- Loading Dataset in the Standard XML Format
- Handling Existing Data
- Other TransX Utility Features

Simplified Multilingual Data Loading

Traditionally, the typical translation data loading method was to switch the `NLS_LANG` setting when you switch files to be loaded. Each of the load files is encoded in a particular character set suitable for the particular language. This was required because translations must be done 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. The TransX Utility loading tool frees the development and translation groups maintaining the correct character set throughout the process until they successfully load the data into the database using XML.

Simplified Data Format Support and Interface

The TransX Utility data loading tool complies with a data format defined to be the canonical method for the representation of any seed data to be loaded to the database. The format is intuitive and easy to understand. The format is also simplified for translation groups to use. The format specification defines how translators can describe the data to load it in the expected way.

The data loading tool has a command-line interface and programmable API. Both of them are straightforward and require little time to learn.

Loading Dataset in the Standard XML Format

Given the dataset (the input data) in the canonical format, the TransX Utility loads the data into the designated locations in the database. It does not, however, create objects, including the table that the data is going to be loaded to. In addition to literal values represented in XML, the following expressions can be used to describe the data to be loaded:

Constant Expression A constant expression allows you to specify a constant value. A column with a fixed value for each row does not have to repeat the same value.

Sequence A column can be loaded with a value obtained from a sequence in the database.

Query A SQL query can be used to load a column. A query can use parameters.

Handling Existing Data

The data loading tool determines whether there are duplicate rows in the database. It also lets you choose how it processes duplicate rows from one of the options in the following list. A row is considered duplicate if the values of all columns specified as lookup-key are the same. The processing options are:

- Skip the duplicate rows or leave them as they are (default)
- Update or overwrite the duplicate rows with the data in provided dataset
- Display an error

Other TransX Utility Features

The lists describes other TransX Utility features:

- Command-line Interface—The data loading tool provides easy-to-use commands.
- User API—The data loading tool exposes a Java API.
- Validation—The data loading tool validates the data format and reports errors.
- White Space Handling—White space characters in the dataset are not significant, unless otherwise specified in various granularity.
- Unloading—Based on a query, the data loading tool exports the result into the standard data format.
- Intimacy with Translation Exchange Format—Designed for transformation to and from translation exchange format
- Localized User Interface—Messages are provided in many languages.

Installing TransX Utility

Here is how to install TransX, and the dependencies of TransX.

Dependencies of TransX

The Oracle TransX utility needs the following components in order to function:

- Database connectivity -- JDBC drivers. The utility can work with any JDBC drivers but is optimized for Oracle's JDBC drivers. Oracle does not guarantee or provide support for TransX running against non-Oracle databases.
- XML Parser -- Oracle XML Parser, Version 2. The Oracle XML Parser, Version 2, is part of the Oracle database installations, and is also available from the Oracle Technology Network (OTN) Web site.
- XML Schema Processor -- Oracle XML Schema Processor. The Oracle XML Schema Processor is part of the Oracle database installations, downloadable from the Oracle Technology Network (OTN) Web site.
- XML SQL Utility-- Oracle XML SQL Utility (XSU). The Oracle XSU is part of the Oracle database installation, and is also available from Oracle Technology Network (OTN) Web site.

Installing TransX Using the Oracle Installer

TransX is packaged with Oracle database. The TransX utility is made up of three executable files:

- `$ORACLE_HOME/rdbms/jlib/transx.zip` -- contains all the java classes which make up TransX `$ORACLE_HOME/rdbms/bin/transx` -- a shell script to invoke TransX from the UNIX command line.
- `$ORACLE_HOME\rdbms\bin\transx.bat` -- a batch file to invoke TransX from the Windows command line.

By default, the Oracle installer installs TransX on your hard drive in the locations specified above.

Installing TransX Downloaded from OTN

Download the correct XDK Java components distribution archive from the Oracle Technology Network. Expand the downloaded archive. Depending on the usage scenario, perform the following install tasks:

To Use the TransX Front-end or Its Java API:

Set up the environment (that is, set `CLASSPATH`) using the `env.xxx` script (located in the bin directory inside the directory created by extracting the XDK download archive):

UNIX users: make sure that the path names in `env.csh` are correct and then enter:

```
source env.csh
```

If you are using a shell other than `csh` or `tcsh`, you will have to edit the file to use your shell's syntax.

Windows users: make sure that the path names in `env.bat` are correct; execute the file.

TransX Utility Command-Line Syntax

The following describes the command-line syntax for the TransX Utility.

```
java oracle.xml.transx.loader [options] connect_string username password
datasource [datasource]
java oracle.xml.transx.loader -v datasource [datasource]
java oracle.xml.transx.loader -x connect_string username password table [column]
java oracle.xml.transx.loader -s connect_string username password filename table
[column]
```

TransX Utility Command-Line Examples

The following are command-line examples for the TransX Utility:

```
java oracle.xml.transx.loader "dlsun9999:1521:mydb" scott tiger foo.xml
java oracle.xml.transx.loader "jdbc:oracle:oci:@mydb" scott tiger foo.xml
java oracle.xml.transx.loader -v foo.xml
java oracle.xml.transx.loader -x "dlsun9999:1521:mydb" scott tiger emp
java oracle.xml.transx.loader -s "dlsun9999:1521:mydb" scott tiger emp.xml emp
ename job
```

TransX Utility Command-line Parameters

Table 12–1 shows the command-line parameters.

Table 12–1 TransX Utility Command-line Parameters

Parameter	Meaning
`connect_string`	JDBC connect string You can omit the connect string information through the '@' symbol. 'jdbc:oracle:thin:@' will be supplied.
`username`	Database user name.
`password`	Password for the database user name.
`datasource`	An XML data source.
`option`	Options in Table 12–2, " TransX Utility Command-line Options".

TransX Utility Command-line Options

Table 12–2 TransX Utility Command-line Options

Option	Meaning	Description
`-u`	Update existing rows.	When this option is specified, existing rows are not skipped but updated. To exclude a column from the update operation, specify the `useforupdate` attribute to be "no".
`-e`	Raise exception if a row is already existing in the database.	When this option is specified, an exception will be thrown if a duplicate row is found. By default, duplicate rows are simply skipped. Rows are considered duplicate if the values for lookup-key column(s) in the database and the dataset are the same.
`-x`	Print data in the database in the predefined format.	Similar to the -s option, it causes TransX to perform the opposite operation of loading. Unlike the -s option, it prints the output to `stdout`. Note: Redirecting this output to a file is discouraged, because intervention of the operating system may result in data loss due to unexpected transcoding.
-s	Save data in the database into a file in the predefined format.	This is an option to perform unloading. It queries the database, formats the result into the predefined XML format and stores it under the specified file name.
-p	Print the XML to load.	Prints out the dataset for insert in the canonical format of XSU.
-t	Print the XML for update.	Prints out the dataset for update in the canonical format of XSU.
-o	Omit validation (as the dataset is parsed it is validated by default).	Causes TransX to skip the format validation, which is performed by default.
-v	Validate the data format and exit without loading.	Causes TransX to perform validation and exit.
-w	Preserve white space.	Causes TransX to treat whitespace characters (such as `\t`, `\r`, `\n`, and ' ') as significant. Consecutive whitespace characters in string data elements are condensed into one space character by default.

Command-line Option Exceptions The following are the command-line option exceptions:

- `-u` and `-e` are mutually exclusive
- `-v` must be the only option followed by data, as in the examples
- `-x` must be the only option followed by connect info and SQL query as in the examples

Omitting all arguments will result in the display of the front-end usage information shown in the table.

> **See Also:** Oracle XML API Reference for complete details of the Java API for TransX Utility

Sample Code for TransX Utility

The following is sample code for the TransX Utility:

```
String  datasrc[] = {"data1.xml", "data2.xml", "data3.xml"};

// instantiate a loader
TransX  transx = loader.getLoader();

// start a data loading session
transx.open( jdbc_con_str, usr, pwd );

// specify operation modes
transx.setLoadingMode( LoadingMode.SKIP_DUPLICATES );
transx.setValidationMode( false );

// load the dataset(s)
for ( int i = 0 ; i < datasrc.length ; i++ )
{
transx.load( datasrc[i] );
}

// cleanup
transx.close();
```

13

Getting Started with XDK C Components

This chapter contains these topics:

- Specifications of XDK C/C++ Components
- Globalization Support for the C XDK Components

Specifications of XDK C/C++ Components

Oracle XDK C/C++ components are built on W3C recommendations. The list of supported standards for release 10.1 are:

- XML 1.0 (Second Edition)
- DOM Level 2.0 Specifications
  - DOM Level 2.0 Core
  - DOM Level 2.0 Traversal and Range
- SAX 2.0 and SAX Extensions
- XSLT/XPath Specifications
  - XSL Transformations (XSLT) 1.0
  - XML Path Language (XPath) 1.0
- XML Schema Specifications
  - XML Schema Part 0: Primer
  - XML Schema Part 1: Structures
  - XML Schema Part 2: Datatypes

What Are the XDK C Components

XDK C components are the basic building blocks for reading, manipulating, transforming, and validating XML documents. Oracle XDK C components consist of the following:

- XML Parser for C: checks if an XML document is well-formed, and optionally validates it against a DTD. The parser constructs an object tree which can be accessed via a DOM interface or operates serially via a SAX interface.
- XSLT Processor for C: provides the ability to format an XML document according to a stylesheet bundled with the parser.
- XVM: high performance XSLT transformation engine.

- XML Schema Processor for C: supports parsing and validating XML files against an XML Schema definition file.

  See Also: "Using the XML Parser for C" on page 14-9 for further discussion of the XDK C components.

Installing the C Components of XDK

If you have installed Oracle Database or Oracle Application Server, then you already have the XDK C components installed. You can also download the latest versions of XDK C components from OTN by following these steps:

1. Navigate to `http://www.oracle.com/technology/tech/xml/`.
2. Click the Software link in the right-hand bar.
3. Logon with your OTN username and password (registration is free if you don't already have an account).
4. Select the Windows or UNIX version to download.
5. Accept all conditions in the licensing agreement.
6. Click the appropriate `*.tar.gz` or `*.zip` file.
7. Extract the files in the distribution:
   a. Choose a directory under which you would like the xdk directory and subdirectories to go.
   b. Change to that directory; then extract the XDK download archive file using:

```
UNIX: tar xvfz xdk_xxx.tar.gz
Windows: use WinZip visual archive extraction tool
```

Setting the UNIX Environment

After installing the UNIX version of XDK, the directory structure is:

```
-$XDK_HOME
     | - bin: executable files
     | - lib: library files
     | - nls/data: Globalization Support data files(*.nlb)
     | - xdk
          | - demo/c: demonstration code
          | - doc/c: documentation
          | - public: header files
          | - mesg: message files (*.msb)
```

Here are all the libraries that come with the UNIX version of XDK C components:

Table 13–1 XDK C Components Libraries

Component	Library	Notes
XML Parser XSLT Processor	libxml10.a	XML Parser for C, which includes DOM, SAX, and XSLT APIs
XML Schema Processor		XML Schema Processor for C

The XDK C components (UNIX) depend on the Oracle CORE and Globalization Support libraries in the following table:

Table 13–2 Dependent Libraries of XDK C Components on UNIX

Component	Library	Notes
CORE Library	libcore10.a	Oracle CORE library
Globalization Support Library	libnls10.a	Oracle Globalization Support common library
	libunls10.a	Oracle Globalization Support library for Unicode support

Command Line Environment Setup

The parser may be called as an executable by invoking bin/xml, which has the following options:

Table 13–3 Parser Command Line Options

Option	Meaning
`-c`	Conformance check only, no validation
`-e` *`encoding`*	Specify default input file encoding ("incoding")
`-E` *`encoding`*	Specify DOM/SAX encoding ("outcoding")
`-f` *`file`*	File - Interpret as filespec, not URI
`-h`	Help - show usage help and full list of flags
`-i` *`n`*	Number of times to iterate the XSLT processing
`-l` *`language`*	Language for error reporting
`-n`	Traverse DOM and report number of elements
`-o` *`XSLoutfile`*	Specify output file of XSLT processor
`-p`	Print document after parsing
`-r`	Do not ignore `<xsl:output>` instruction in XSLT processing
`-s` *`stylesheet`*	Style sheet - specifies the XSL style sheet
`-v`	Version - display parser version and then exit
`-V` *`var value`*	To test top level variables in CXSLT
`-w`	Whitespace - preserve all whitespace
`-W`	Warning - stop parsing after a warning
`-x`	SAX - exercise SAX interface and print document

Check if the environment variable `ORA_NLS10` is set to point to the location of the Globalization Support data files. If you install the Oracle database, you can set it to be:

```
setenv ORA_NLS10 ${ORACLE_HOME}/nls/data
```

If no Oracle database is installed, you can use the Globalization Support data files that come with the XDK release by setting:

```
setenv ORA_NLS10 ${XDK_HOME}/nls/data
```

Error message files are provided in the `mesg` subdirectory. Files ending in `.msb` are machine-readable and needed at runtime; files ending in `.msg` are human-readable and contain cause and action descriptions for each error. The messages files also exist in the `$ORACLE_HOME/xdk/mesg` directory.

If you do not have an ORACLE_HOME, check if the environment variable `ORA_XML_MESG` is set to point to the absolute path of the mesg directory. If the Oracle database is installed, you can set `ORA_XML_MESG`, although this is not required:

```
setenv ORA_XML_MESG ${ORACLE_HOME}/xdk/mesg
```

If no Oracle database is installed, you must set the environment variable `ORA_XML_MESG` to point to the absolute path of the `mesg` subdirectory:

```
setenv ORA_XML_MESG ${XDK_HOME}/xdk/mesg
```

The parser may also be invoked by writing code to use 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 `demo` subdirectory for full details of how to build your program.

To get the XDK version you are using on UNIX:

```
strings libxml10.a | grep -i Version
```

Setting the Windows Environment

These are the Windows libraries that come with the XDK C components:

Table 13–4 XDK C Components Libraries on Windows

Component	Library	Notes
XML Parser XSL Processor XML Schema Processor	oraxml10.lib oraxml10.dll	XML Parser for C, which includes DOM, SAX, and XSLT APIs XML Schema Processor for C

The XDK C components (Windows) depend on the Oracle CORE and Globalization Support libraries in the following table:

Table 13–5 Dependent Libraries of XDK C Components on Windows

Component	Library	Notes
CORE Library	oracore10.dll	Oracle CORE library
Globalization Support Library	oranls10.dll	Oracle Globalization Support common library
Globalization Support Library	oraunls10.dll	Oracle Globalization Support library for Unicode support

Environment for Command Line Usage

For the parser and schema validator options, see Table 13–3, " Parser Command Line Options".

Check that the environment variable `ORA_NLS10` is set to point to the location of the Globalization Support encoding definition files. You can set it this way:

```
setenv ORA_NLS10 %ORACLE_HOME%\nls\data
```

If no Oracle database is installed, you can use the Globalization Support encoding definition files that come with the XDK release (a subset of which are in the Oracle database):

```
set ORA_NLS10 =%XDK_HOME%\nls\data
```

Error message files are provided in the `mesg` subdirectory. Files ending in `.msb` are machine-readable and needed at runtime; files ending in `.msg` are human-readable and include cause and action descriptions for each error. The messages files also exist in the `$ORACLE_HOME/xdk/mesg` directory.

If there is an Oracle database installed, you can set `ORA_XML_MESG`, although this is not required:

```
set ORA_XML_MESG =%ORACLE_HOME%\xdk\mesg
```

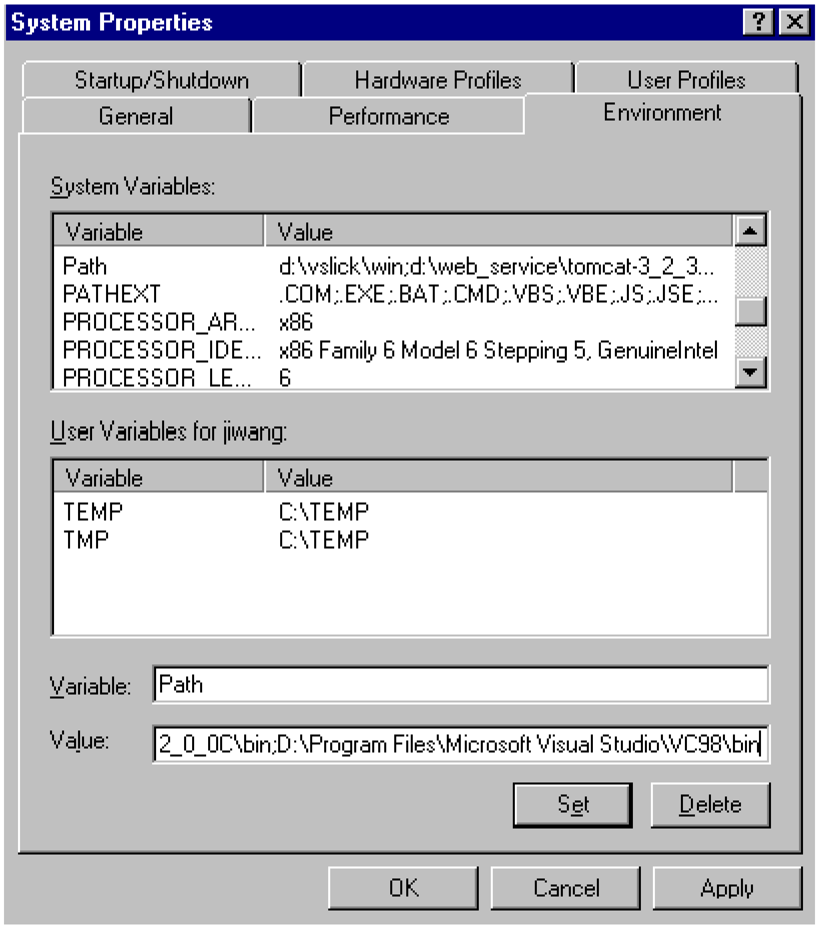
If no Oracle database is installed, you must set the environment variable `ORA_XML_MESG` to point to the absolute path of the `mesg` subdirectory:

```
set ORA_XML_MESG =%XDK_HOME%\xdk\mesg
```

In order to compile the sample code, you set the path for the `cl` compiler.

Go to the Start Menu and select Settings > Control Panel. In the pop-up window of Control Panel, select System icon and double click. A window named System Properties pops up. Select Environment Tab and input the path of `cl.exe` to the PATH variable shown in Figure 13–1, "Setting the Path for the cl Compiler in Windows".

Figure 13–1 Setting the Path for the cl Compiler in Windows



System Variables:

Variable	Value
Path	d:\vslick\win;d:\web_service\tomcat-3_2_3...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;...
PROCESSOR_AR...	x86
PROCESSOR_IDE...	x86 Family 6 Model 6 Stepping 5, GenuineIntel
PROCESSOR_LE...	6

User Variables for jiwang:

Variable	Value
TEMP	C:\TEMP
TMP	C:\TEMP

Variable: Path

Value: 2_0_0C\bin;D:\Program Files\Microsoft Visual Studio\VC98\bin

You need to update the `Make.bat` by adding the path of the libraries and the header files to the compile and link commands as shown in the following example of a `Make.bat` file:

```
: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 -
ID:\Progra~1\Micros~1\VC98\Include %filename%.c
goto :EOF

:LINK
set filename=%1
link %link_dbg% /out:..\..\..\..\bin\%filename%.exe /libpath:%ORACLE_HOME%\lib
/libpath:D:\Progra~1\Micros~1\VC98\lib /libpath:..\..\..\..\lib %filename%.obj
oraxml10.lib oracore10.lib oranls10.lib oraunls10.lib user32.lib kernel32.lib
msvcrt.lib ADVAPI32.lib oldnames.lib winmm.lib
:EOF
```

where:

`D:\Progra~1\Micros~1\VC98\Include:` is the path for header files and `D:\Progra~1\Micros~1\VC98\lib:` is the path for library files.

Using the XDK C Components with Visual C++

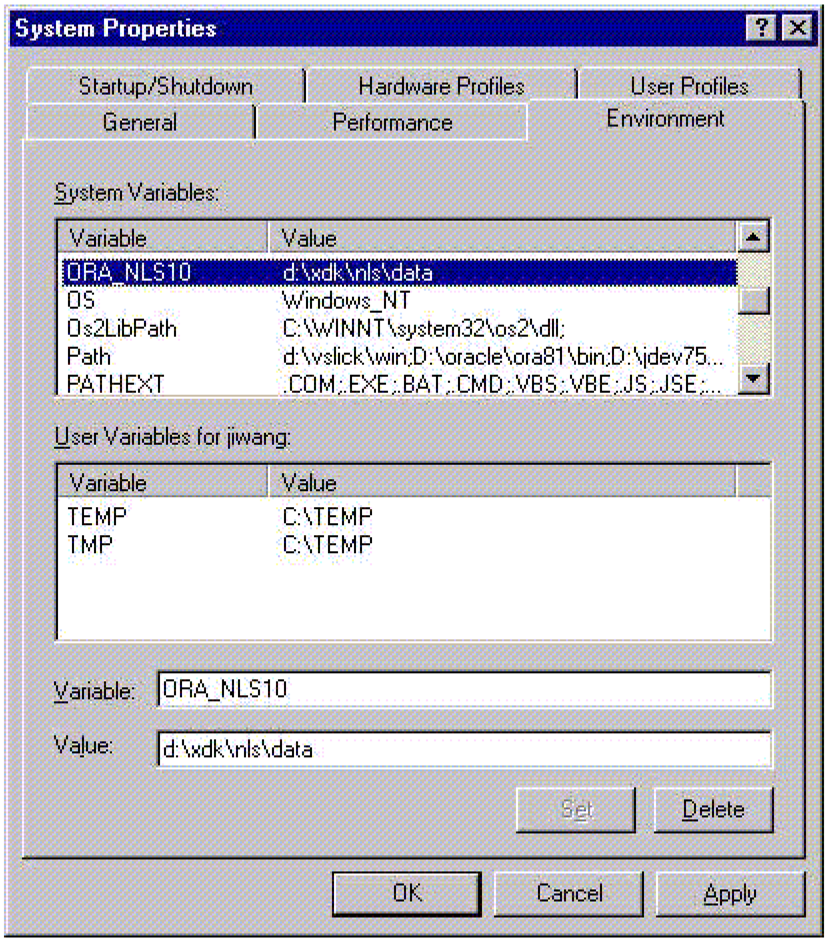
If you are using Microsoft Visual C++ compiler:

Check that the environment variable `ORA_NLS10` is set to point to the location of the Globalization Support data files.

In order to use Visual C++, you need to employ the system setup for Windows to define the environment variable.

Go to Start Menu and select Settings > Control Panel. In the pop up window of Control Panel, select System icon and double click. A window named System Properties pops up. Select Environment Tab and input `ORA_NLS10`, and its value `d:\xdk\nls\data`, as shown in Figure 13–2:

Figure 13–2 Setting Up the ORA_NLS10 Environment Variable



System Variables:

Variable	Value
ORA_NLS10	d:\xdk\nls\data
OS	Windows_NT
Os2LibPath	C:\WINNT\system32\os2\dll;
Path	d:\vslick\win;D:\oracle\ora81\bin;D:\jdev75...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;...

User Variables for jiwang:

Variable	Value
TEMP	C:\TEMP
TMP	C:\TEMP

Variable: ORA_NLS10

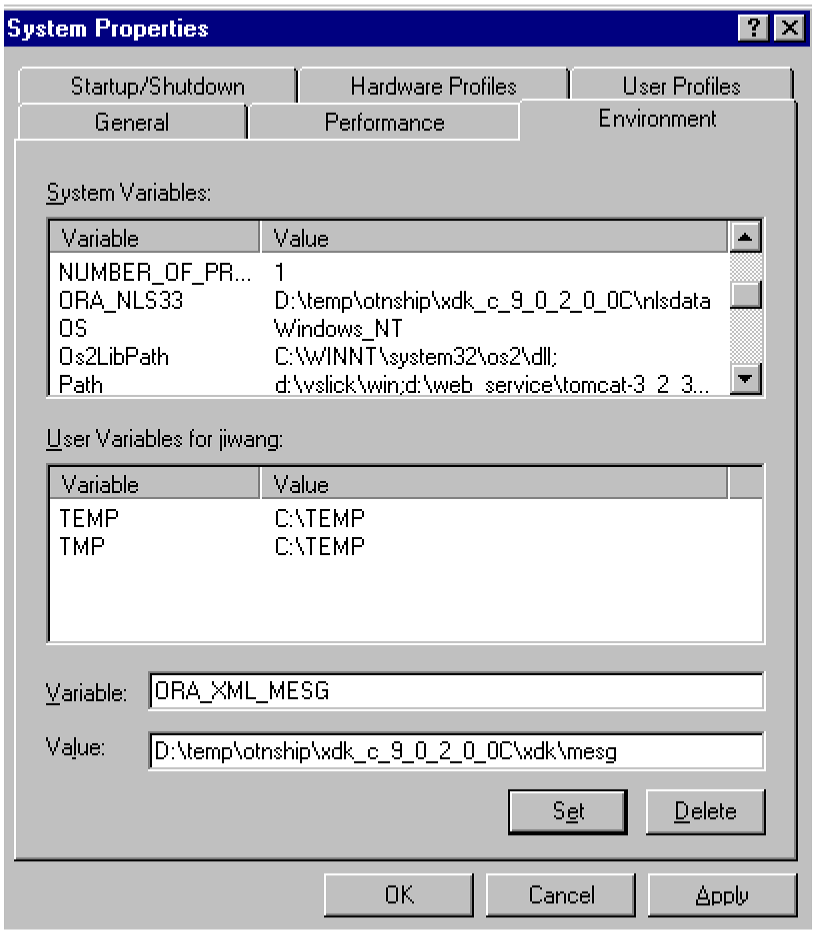
Value: d:\xdk\nls\data

Check that the environment variable `ORA_XML_MESG` is set to point to the absolute path of the `mesg` directory.

In order for Visual C++ to use the environment variable, you need to employ the system setup for Windows to define the environment variable.

Go to the Start Menu and select Settings > Control Panel. In the pop-up window of Control Panel, select System icon and double click. A window named System Properties pops up. Select Environment Tab and input `ORA_XML_MESG`, as in Figure 13–3, (the illustrations show screens for a previous release).

Figure 13–3 Setting Up the ORA_XML_MESG Environment Variable



System Variables:

Variable	Value
NUMBER_OF_PR...	1
ORA_NLS33	D:\temp\otnship\xdk_c_9_0_2_0_0C\nlsdata
OS	Windows_NT
Os2LibPath	C:\WINNT\system32\os2\dll;
Path	d:\vslick\win;d:\web_service\tomcat-3_2_3...

User Variables for jiwang:

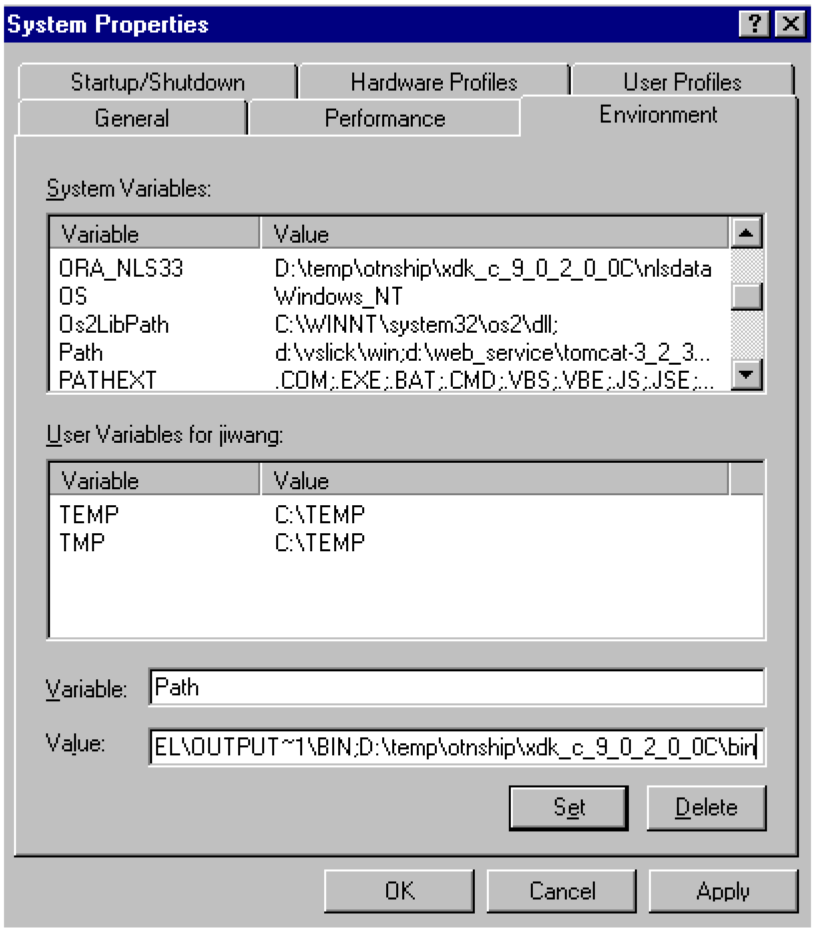
Variable	Value
TEMP	C:\TEMP
TMP	C:\TEMP

Variable: ORA_XML_MESG

Value: D:\temp\otnship\xdk_c_9_0_2_0_0C\xdk\mesg

Figure 13–4 shows the setup of the PATH for DLLs:

Figure 13–4 Setup of the PATH for DLLs



System Variables:

Variable	Value
ORA_NLS33	D:\temp\otnship\xdk_c_9_0_2_0_0C\nlsdata
OS	Windows_NT
Os2LibPath	C:\WINNT\system32\os2\dll;
Path	d:\vslick\win;d:\web_service\tomcat-3_2_3...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;...

User Variables for jiwang:

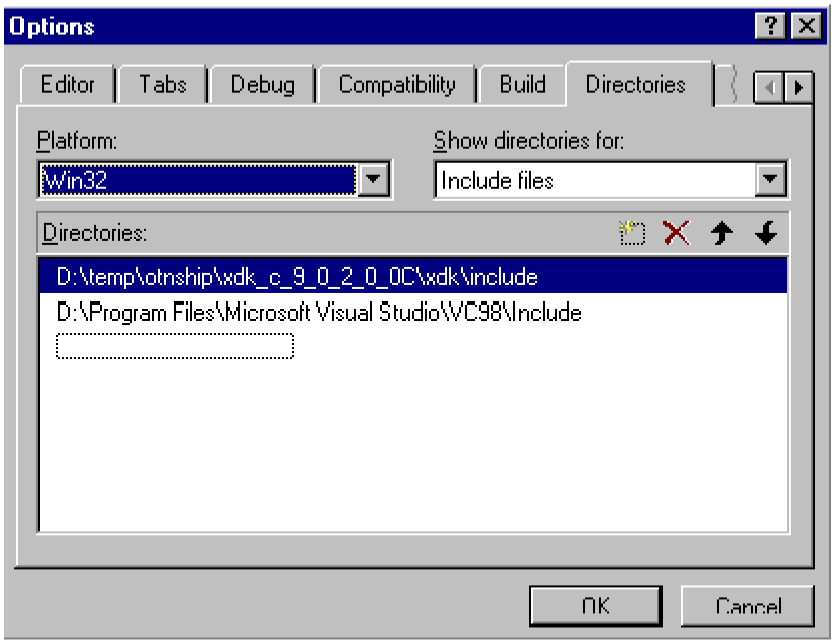
Variable	Value
TEMP	C:\TEMP
TMP	C:\TEMP

Variable: Path

Value: EL\OUTPUT~1\BIN;D:\temp\otnship\xdk_c_9_0_2_0_0C\bin

After you open a workspace in Visual C++ and include the `*.c` files for your project, you must set the path for the project. Go to the Tools menu and select Options. A window will pop up. Select the Directory tab and set your include path as shown in Figure 13–5:

Figure 13–5 Setting Your Include Path in Visual C++



Platform: Win32

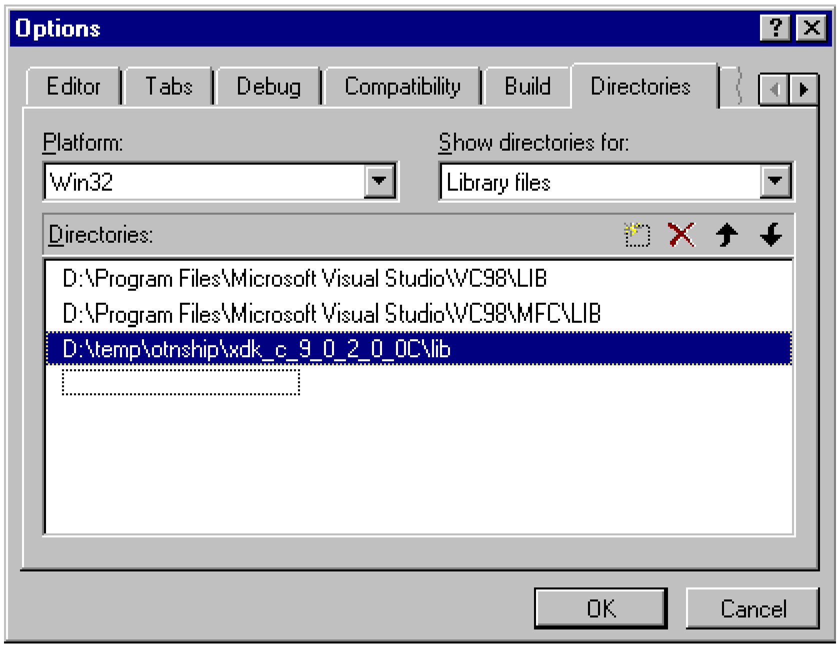
Show directories for: Include files

Directories:

D:\temp\otnship\xdk_c_9_0_2_0_0C\xdk\include
D:\Program Files\Microsoft Visual Studio\VC98\Include

Then set your library path as shown in Figure 13–6:

Figure 13–6 Setting Your Static Library Path in Visual C++



Platform: Win32

Show directories for: Library files

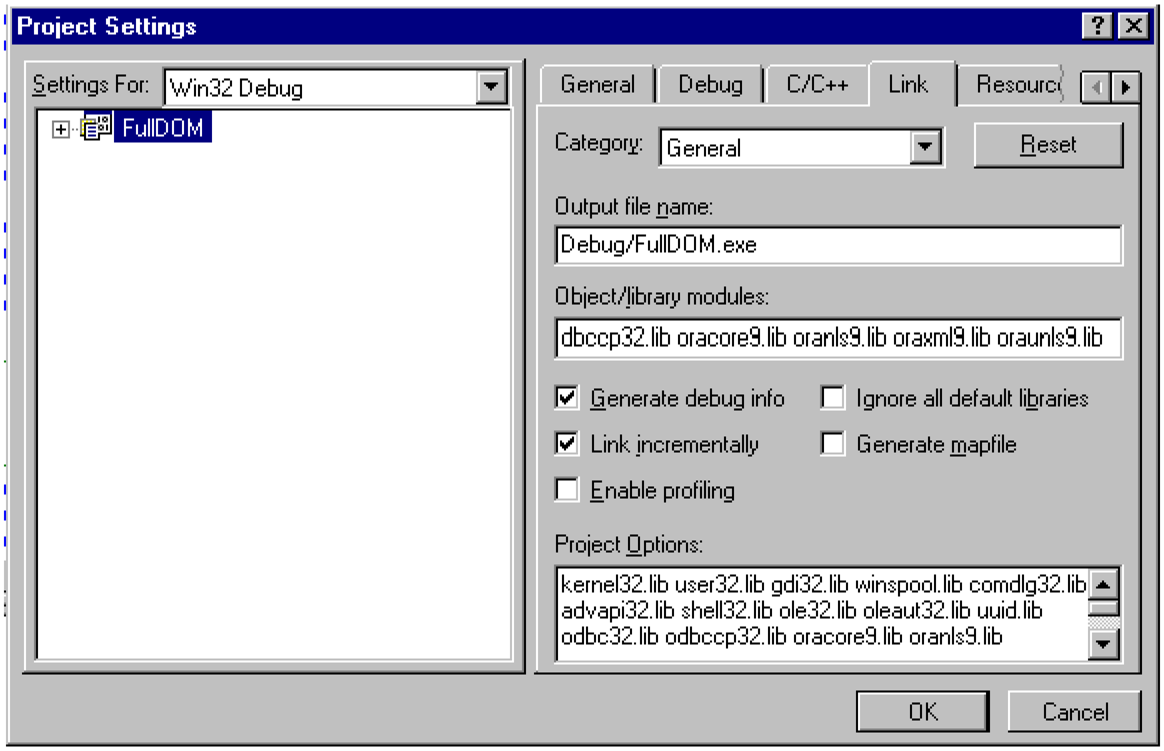
Directories:

D:\Program Files\Microsoft Visual Studio\VC98\LIB
D:\Program Files\Microsoft Visual Studio\VC98\MFC\LIB
D:\temp\otnship\xdk_c_9_0_2_0_0C\lib

After setting the paths for the static libraries in `%XDK_HOME%\lib`, you also need to set the library name in the compiling environment of Visual C++.

Go to the Project menu in the menu bar and select Settings. A window pops up. Please select the Link tab in the Object/Library Modules field enter the name of XDK C components libraries, as shown in Figure 13–7:

Figure 13–7 Setting Up the Static Libraries in Visual C++ Project



Settings For: Win32 Debug

FullDOM

Category: General

Output file name: Debug/FullDOM.exe

Object/library modules: dbccp32.lib oracore9.lib oranls9.lib oraxml9.lib oraunls9.lib

[✓] Generate debug info
[ ] Ignore all default libraries
[✓] Link incrementally
[ ] Generate mapfile
[ ] Enable profiling

Project Options:
kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib oracore9.lib oranls9.lib

Optionally, compile and run the demo programs. Then you can start using C XDK components.

Globalization Support for the C XDK Components

The parser 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-GB-2312, KOI8-R, KSC5601, EUC-KR, ISO-2022-CN, ISO-2022-JP, ISO-2022-KR, WINDOWS-{1250-1258}, EBCDIC-CP-{US,CA,NL,WT,DK,NO,FI,SE,IT,ES,GB,FR,HE,BE,CH,ROECE,YU,IS,AR1} , IBM{037,273,277,278,280,284,285,297,420,424,437,500,775,850,852,855,857,00858, 860,861,863,865,866,869,870,871,1026,01140,01141,01142,01143,01144,01145,01146, 01147,01148}

Any alias of the above character sets that is found here may also be used. In addition, any character set specified in Appendix A, Character Sets, of the *Oracle Database Globalization Support Guide* can be used with the exception of IW7IS960.

However, it is recommended that you use IANA character set names for interoperability with other XML parsers. Also note that XML parsers are only required to support UTF-8 and UTF-16 so those character sets should be preferred.

In order to be able to use these encodings, you should have the ORACLE_HOME environment variable set and pointing to the location of your Oracle installation. This enables the use of the globalization support data files which contain data for all supported encodings. On UNIX systems, they are usually in `$ORACLE_HOME/nls/data`. On Windows, they are usually in `%ORACLE_HOME%\nls\data`. C and C++ XDK releases that are downloaded from OTN contain an `nls/data` subdirectory. You must set the environment variable ORA_NLS10 to the absolute path of the `nls/data` subdirectory if you do not have an Oracle installation.

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) since single-byte performance is by far the fastest. The flag `XML_FLAG_FORCE_INCODING` says that the default input encoding should always be applied to input documents, ignoring any BOM or XMLDecl. However, a protocol declaration (such as HTTP character set) is always honored.

The data encoding for DOM and SAX ("outcoding") should be chosen 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, an error occurs. So for maximum utility, a Unicode-based outcoding should be used, since Unicode can represent any character. If outcoding is not specified, it defaults to the incoding of the first document parsed.

14 XML Parser for C

This chapter contains these topics:

What Is the Unified C API for XDK and Oracle XML DB?

The single DOM is part of the unified C API, which is a C **API** for XML, whether the XML is in the database or in documents outside the database. DOM means DOM 2.0 plus non-standard extensions in XDK for XML documents or for Oracle XML DB for XML stored as an `XMLType` column in a table, usually for performance improvements.

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 unified C API is a programming interface that includes the union of all functionality needed by XDK and Oracle XML DB, with XSLT and XML Schema as primary customers. The DOM 2.0 standard was followed as closely as possible, though some naming changes were required when mapping from the objected-oriented DOM specification to the flat C namespace (overloaded `getName()` methods changed to `getAttrName()` and so on).

Unification of the functions is accomplished by conforming contexts: a top-level XML context (`xmlctx`) intended to share common information between cooperating XML components. Data encoding, error message language, low-level memory allocation callbacks, and so on, are defined here. This information is needed before a document can be parsed and DOM or SAX output.

Both the XDK and the Oracle XML DB need different startup and tear-down functions for both contexts (top-level and service). The initialization function takes implementation-specific arguments and returns a conforming context. A conforming context means that the returned context must begin with a `xmlctx`; it may have any additional implementation-specific parts following that standard header.

Initialization (getting an `xmlctx`) is an implementation-specific step. Once that `xmlctx` has been obtained, unified DOM calls are used, all of which take an `xmlctx` as the first argument.

This interface (new for release 10.1) supersedes the existing C API. In particular, the `oraxml` interfaces (top-level, DOM, SAX and XSLT) and `oraxsd` (Schema) interfaces are deprecated.

Using DOM for XDK

When the XML resides in a traditional file system, or the Web, or something similar, the XDK package is used. Again, only for startup are there any implementation-specific steps.

First a top-level `xmlctx` is needed. This contains encoding information, low-level memory callbacks, error message language, and encoding, and so on (in short, those things which should remain consistent for all XDK components). An `xmlctx` is allocated with `XmlCreate()`.

```
xmlctx *xctx;
xmlerr  err;

xctx = (xmlctx *) XmlCreate(&err, "xdk context", "data-encoding", "ascii", ...,
NULL);
```

Once the high-level XML context has been obtained, documents may be loaded and DOM events generated. To generate DOM:

```
xmldocnode *domctx;
xmlerr err;

domctx = XmlLoadDom(xctx, &err, "file", "foo.xml", NULL);
```

To generate SAX events, a SAX callback structure is needed:

```
xmlsaxcb saxcb = {
  UserAttrDeclNotify,            /* user's own callback functions */
  UserCDATANotify,
  ...
  };

if (XmlLoadSax(xctx, &saxcb, NULL, "file", "foo.xml", NULL) != 0)
  /*    an error occured  */
```

The tear-down function for an XML context, `xmlctx,` is `XmlDestroy()`.

Loading an XML Document with the C API

Once an `xmlctx` is obtained, a serialized XML document is loaded with the `XmlLoadDom()` or `XmlLoadSax()` functions. Given the Document node, all API DOM functions are available.

Data Encoding of XML Documents for the C API

XML data occurs in many encodings. You have control over the encoding in three 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 data is always in some encoding. Some encodings are entirely self-describing, such as UTF-16, which requires a specific BOM before the start of the actual data. A document's encoding may also be specified in the `XMLDecl` or MIME header. If the specific encoding cannot be determined, your default input encoding is applied. If no default is provided by you, UTF-8 is assumed on ASCII platforms and UTF-E on EBCDIC platforms.

A provision is made for cases when the encoding information of the input document is corrupt. For example, if an ASCII document which contains an `XMLDecl` saying `encoding=ascii` is blindly converted to EBCDIC, the new EBCDIC document contains (in EBCDIC), an `XMLDecl` which claims the document is ASCII, when it is not. The correct behavior for a program which is re-encoding XML data is to regenerate the XMLDecl, not to convert it. The `XMLDecl` is metadata, not data itself. However, this rule is often ignored, and then the corrupt documents result. To work around this problem, an additional flag is provided which allows the input encoding to be forcibly set, overcoming an incorrect `XMLDecl`.

The precedence rules for determining input encoding are as follows:

1. Forced encoding as specified by the user.

> **Caution:** This 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. Then the parser will object about the conflict.

2. Protocol specification (HTTP header, and so on).
3. `XMLDecl` specification is used.
4. User's default input encoding.
5. The default: UTF-8 (ASCII platforms) or UTF-E (EBCDIC platforms).

Once the input encoding has been determined, the document can be parsed and the data presented. You are allowed to choose the presentation encoding; the data will be in that encoding regardless of the original input encoding.

When a DOM is written back out (serialized), you can choose at that time to re-encode the presentation data, and the final serialized document can be in any encoding.

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. However, Oracle XML DB data when stored in table form, is *not* NULL-terminated but *length-encoded*, so an additional set of length-encoded APIs are provided for the high-frequency cases to improve performance (if you deliberately choose to use them). Either set of functions works.

In particular, the following DOM functions are invoked frequently and have dual APIs:

Table 14–1 NULL-Terminated and Length-Encoded C API Functions

NULL-Terminated API	Length-Encoded API
`XmlDomGetNodeName()`	`XmlDomGetNodeNameLen()`
`XmlDomGetNodeLocal()`	`XmlDomGetNodeLocalLen()`
`XmlDomGetNodeURI()`	`XmlDomGetNodeURILen()`
`XmlDomGetNodeValue()`	`XmlDomGetNodeValueLen()`
`XmlDomGetAttrName()`	`XmlDomGetAttrNameLen()`
`XmlDomGetAttrLocal()`	`XmlDomGetAttrLocalLen()`
`XmlDomGetAttrURI()`	`XmlDomGetAttrURILen()`
`XmlDomGetAttrValue()`	`XmlDomGetAttrValueLen()`

Error Handling

The 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, error codes are stored and the last error can be retrieved with `XmlDomGetLastError()`.

Error messages, by default, are output to `stderr`. However, you can register an error message callback at initialization time. When an error occurs, that callback will be invoked and no error printed.

Installing the C API

There are no special installation or first-use requirements. The XML DOM does not require an ORACLE_HOME. It can run out of a reduced root directory such as those provided on OTN releases.

However, since the XML DOM requires globalization support, the globalization support data files must be present (and found through the environment variables ORACLE_HOME or ORA_NLS10).

Using OCI and the C API

The C API for XML can be used for `XMLType` columns in the database. XML data that is stored in a database table can be accessed in an Oracle Call Interface (OCI) program by initializing the values of OCI handles, such as environment handle, service handle, error handle, and optional parameters. These input values are passed to the function `OCIXmlDbInitXmlCtx()` and an XML context is returned. After the calls to the C API are made, the context is freed by the function `OCIXmlDbFreeXmlCtx()`.

XML Context

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 as their prototypes:

```
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*, "OCI XML DB Functions" for reference material on the functions.
- *Oracle Call Interface Programmer's Guide*, "OCI Support for XML" for a discussion about OCI support for XML.
- *Oracle XML API Reference*, "DOM APIs for C".

Creating XMLType Instances on the Client

New `XMLType` instances on the client can be constructed using the `XmlLoadDom()` calls. You first have to initialize the `xmlctx`, as in the example in Using DOM for XDK on page 14-2. The XML data itself can be constructed from a user buffer, local file, or URI. The return value from these is an `(xmldocnode *)` which can be used in the rest of the common C API. Finally, the `(xmldocnode *)` can be cast to a `(void *)` and directly provided as the bind value if required.

Empty `XMLType` instances can be constructed using the `XmlCreateDocument()` call. This would be equivalent to an `OCIObjectNew()` for other types. You can operate on the `(xmldocnode *)` returned by the above call and finally cast it to a `(void *)` if it needs to be provided as a bind value.

XML Data on the Server

XML data on the server can be operated on by means of OCI statement calls. You can bind and define `XMLType` values using `xmldocnode`, as with other object instances. OCI statements are used to select XML data from the server. This data can be used in the C DOM functions directly. Similarly, the values can be bound back to SQL statements directly.

XMLType Functions and Descriptions

The following table describes a few of the functions for XML operations.

Table 14–2 XMLType Functions

Description	Function Name
Create empty `XMLType` instance	`XmlCreateDocument()`
Create from a source buffer	`XmlLoadDom()` and so on
Extract an `XPath` expression	`XmlXPathEvalexpr()` and family
Transform using an XSL stylesheet	`XmlXslProcess()` and family
Check if an `XPath` exists	`XmlXPathEvalexpr()` and family
Is document schema-based?	`XmlDomIsSchemaBased()`
Get schema information	`XmlDomGetSchema()`
Get document namespace	`XmlDomGetNodeURI()`

Table 14–2 (Cont.) XMLType Functions

Description	Function Name
Validate using schema	`XmlSchemaValidate()`
Obtain DOM from `XMLType`	Cast `(void *)` to `(xmldocnode *)`
Obtain `XMLType` from DOM	Cast `(xmldocnode *)` to `(void *)`

OCI Examples

Here is an example of how to construct a schema-based document using the DOM API and save it to the database (you must include the header files `xml.h` and `ocixmldb.h`):

```
#include <xml.h>
#include <ocixmldb.h>
static oratext tlpxml_test_sch[] = "<TOP xmlns='example1.xsd'\n\
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' \n\
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 */
    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,
                    (ub4)strlen((char *)sqlstmt),
                    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT)) {
     return OCI_ERROR;
   }

   if(status = OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 1, (dvoid *) 0,
                   (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)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;
}
```

Here is an example of how to get a document from the database and modify it using the DOM API:

```
#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) 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,(oratext *)"Item");
      ilist_l   = XmlDomGetNodeListLength(xctx, item_list);
      printf(" Item list length = %d \n", 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 the XML Parser for C

The XML Parser for C is provided with the Oracle Database and the Oracle Application Server. It is also available for download from http://www.oracle.com/technology/tech/xml.

It is located in `$ORACLE_HOME/xdk/` on UNIX systems.

`readme.html` in the `doc` directory of the software archive contains release specific information including bug fixes and API additions.

The XML Parser for C checks if an XML document is well-formed, and optionally, validates it against a DTD. The parser constructs an object tree which can be accessed through a DOM interface or the parser operates serially through a SAX interface.

You can post questions, comments, or bug reports to the XML Discussion Forum at http://www.oracle.com/technology/tech/xml.

There are several sources of information on specifications:

See Also:

- *Oracle XML API Reference* "DOM APIs for C"
- *Oracle XML API Reference* "SAX APIs for C"
- *Oracle XML API Reference* "Callback APIs for C"
- *Oracle XML API Reference* "Datatypes for C"
- `http://www.oracle.com/technology/tech/xml/`

Memory Allocation

The memory callback functions XML_ALLOC_F and XML_FREE_F can be used if you want to use your own memory allocation. If they are used, both of the functions should be specified.

The memory allocated for parameters passed to the SAX callbacks or for nodes and data stored with the DOM parse tree are not freed until one of the following is done:

- `XmlFreeDocument()` is called.
- `XmlDestroy()` is called.

Thread Safety

If threads are forked off somewhere in the init-parse-term sequence of calls, you get unpredictable behavior and results.

Data Types Index

Table 14–3 lists the datatypes used in XML Parser for C.

Table 14–3 Datatypes Used in XML Parser for C

Datatype	Description
`oratext`	String pointer
`xmlctx`	Master XML context
`xmlsaxcb`	SAX callback structure (SAX only)
`ub4`	32-bit (or larger) unsigned integer
`uword`	Native unsigned integer

Error Message Files

Error messages files are 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 not required.

XML Parser for C Calling Sequence

Figure 14–1 describes the XML Parser for C calling sequence as follows:

1. `XmlCreate()` function initializes the parsing process.
2. The parsed item can be an XML document (file) or string buffer. The input is parsed using the `XmlLoadDom()` function.
3. DOM or SAX API:

DOM: If you are using the DOM interface, include the following steps:

- The `XmlLoadDom()` function calls `XmlDomGetDocElem()`.
- This first step calls other DOM functions as required. These other DOM functions are typically node or print functions that output the DOM document.
- You can first invoke `XmlFreeDocument()` to clean up any data structures created during the parse process.

SAX: If you are using the SAX interface, include the following steps:

- Process the results of the parser from `XmlLoadSax()` using callback functions.
- Register the callback functions. Note that any of the SAX callback functions can be set to `NULL` if not needed.

4. Use `XmlFreeDocument()` to clean up the memory and structures used during a parse, and go to Step 5. or return to Step 2.
5. Terminate the parsing process with `XmlDestroy()`

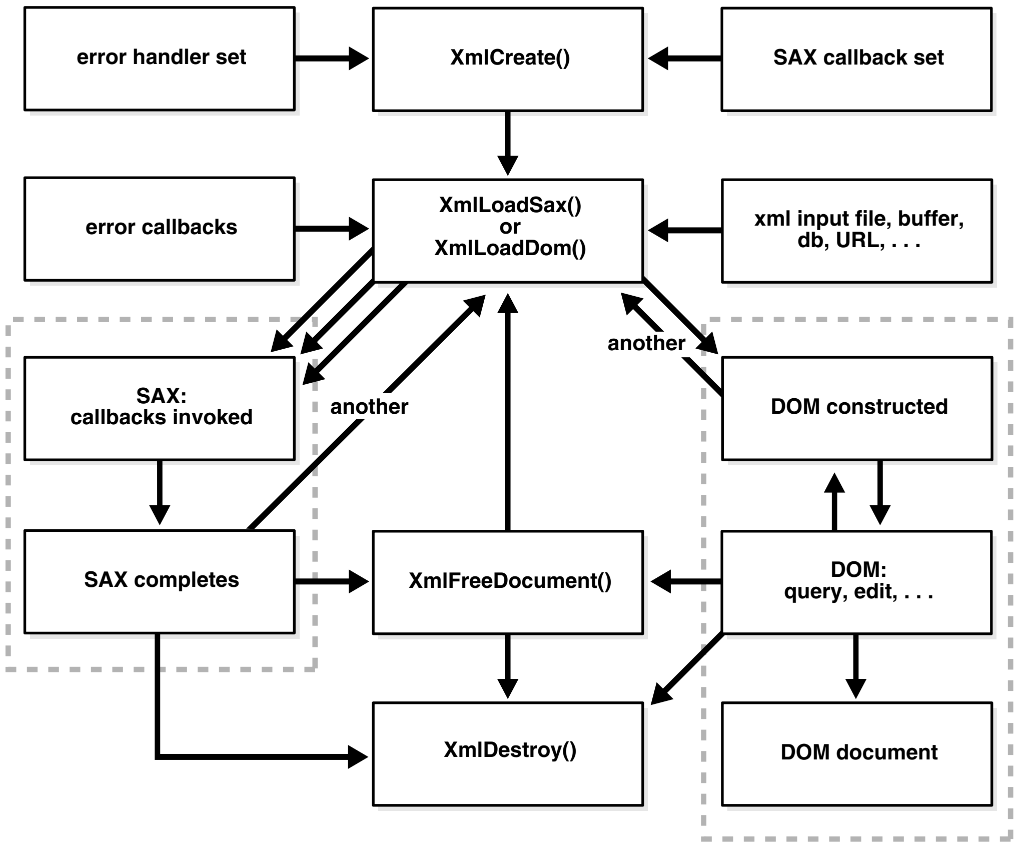
Parser Calling Sequence

The sequence of calls to the parser can be any of the following:

- `XmlCreate() - XmlLoadDom() - XmlDestroy()`
- `XmlCreate() - XmlLoadDom() - XmlFreeDocument() -`

  `XmlLoadDom() - XmlFreeDocument() - ... - XmlDestroy()`
- `XmlCreate() - XmlLoadDom() -... - XmlDestroy()`

Figure 14–1 XML Parser for C Calling Sequence



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 an error handler is provided.
- The default behavior for the parser is to check that the input is well-formed but not to check whether it is valid. The property "validate" can be set to validate the input. The default behavior for whitespace processing is to be fully conforming to the XML 1.0 specification, 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 property "discard-whitespace"which discards 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.

DOM and SAX APIs Compared

Oracle XML parser for C checks if an XML document is well-formed, and optionally validates it against a DTD. The parser constructs an object tree which can be accessed through one of the following interfaces:

- 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.

  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.

- 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 callbacks, 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.

  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.

Using the SAX API

To use SAX, an `xmlsaxcb` structure is initialized with function pointers and passed to the `XmlLoadSax()` call. A pointer to a user-defined context structure can also be included. That context pointer will be passed to each SAX function.

SAX Callback Structure

The SAX callback structure can be found at:

See Also: *Oracle XML API Reference*, SAX

Command Line Usage

The XML Parser and XSLT Processor can be called as an executable by invoking `bin/xml`:

```
xml [options] [document URI]
or
xml -f [options] [document filespec]
```

Table 14–4 lists the command line options.

Table 14–4 XML Parser and XSLT Processor: Command Line Options

Option	Description
`-B` *`BaseUri`*	Set the Base URI for XSLT processor: `BaseUri` of `http://pqr/xsl.txt` resolves `pqr.txt` to `http://pqr/pqr.txt`
`-c`	Conformance check only, no validation.
`-e` *`encoding`*	Specify input file encoding.

Table 14–4 (Cont.) XML Parser and XSLT Processor: Command Line Options

Option	Description
`-E` *`encoding`*	Specify DOM or SAX encoding.
`-f`	File - interpret as filespec, not URI.
`-G` *`xptrexprs`*	Evaluates XPointer schema examples given in a file.
`-h`	Help - show this usage. (-hh for more options.)
`-hh`	Show complete options list.
`-i` *`n`*	Number of times to iterate the XSLT processing.
`-l` *`language`*	Language for error reporting.
`-n`	Number - DOM traverse and report number of elements.
`-o` *`XSLoutfile`*	Specifies output file of XSLT processor.
`-p`	Print document and DTD structures after parse.
`-P`	Pretty print from root element.
`-PE` *`encoding`*	Specifies encoding for -P or -PP output.
`-PP`	Pretty print from root node (DOC); includes XMLDecl.
`-PX`	Include XMLDecl in output always.
`-r`	Do not ignore `<xsl:output>` instruction in XSLT processing.
`-s` *`stylesheet`*	Specifies the XSLT stylesheet.
`-v`	Version - display parser version then exit.
`-V` *`var value`*	Test top-level variables in C XSLT.
`-w`	Whitespace - preserve all whitespace.
`-W`	Warning - stop parsing after a warning.
-x	Exercise SAX interface and print document.

Writing C Code to Use Supplied APIs

XML Parser for C can also be invoked by writing code to use the supplied APIs. The code must be compiled using the headers in the `include/` subdirectory and linked against the libraries in the `lib/` subdirectory. Please see the `Makefile` in the `demo/c/` subdirectory for full details of how to build your program.

Using the Sample Files

The `$ORACLE_HOME/xdk/demo/c/` directory contains several XML applications to illustrate how to use the XML Parser for C with the DOM and SAX interfaces.

To build the sample programs, change directories to the sample directory (`$ORACLE_HOME/xdk/demo/c/` on UNIX) and read the `README` file. This file explains how to build the sample programs.

Table 14–5 lists the sample files:

Table 14–5 XML Parser for C Sample Files

Sample File Name	Description
`DOMNamespace.c`	Source for `DOMNamespace` program.

Table 14–5 (Cont.) XML Parser for C Sample Files

Sample File Name	Description
`DOMNamespace.std`	Expected output from `DOMNamespace`.
`DOMSample.c`	Source for `DOMSample` program.
`DOMSample.std`	Expected output from `DOMSample`.
`FullDOM.c`	Sample usage of DOM interface.
`FullDOM.std`	Expected output from `FullDOM`.
`Makefile`	Batch file for building sample programs.
`NSExample.xml`	Sample XML file using namespaces.
`SAXNamespace.c`	Source for `SAXNamespace` program.
`SAXNamespace.std`	Expected output from `SAXNamespace`.
`SAXSample.c`	Source for `SAXSample` program.
`SAXSample.std`	Expected output from `SAXSample`.
`XSLSample.c`	Source for `XSLSample` program.
`XSLSample.std`	Expected output from `XSLSample`.
`XVMSample.c`	Source for `XVMSample` program.
`XVMSample.std`	Expected output from `XVMSample`.
`XSLXPathSample.c`	Source for `XSLXPathSample` program.
`XSLXPathSample.std`	Expected output from `XSLXPathSample` program.
`XVMXPathSample.c`	Source for `XVMXPathSample` program.
`XVMXPathSample.std`	Expected output from `XVMXPathSample` program.
`class.xml`	XML file that may be used with `XSLSample`.
`iden.xsl`	Stylesheet that may be used with `XSLSample`.
`cleo.xml`	The Tragedy of Antony and Cleopatra, XML version of Shakespeare's play.

Table 14–6 lists the programs built by the sample files:

Table 14–6 XML Parser for C: Sample Built Programs

Built Program	Description
`DOMSample`	A sample application using DOM APIs (shows an outline of Cleopatra, that is, the XML elements ACT and SCENE).
`SAXSample` *word*	A sample application using SAX APIs. Given a word, shows all lines in the play *Cleopatra* containing that word. If no word is specified, 'death' is used.
`DOMNamespace`	Same as `SAXNamespace`, except using DOM interface.
`SAXNamespace`	A sample application using Namespace extensions to SAX API; prints out all elements and attributes of `NSExample.xml` along with full namespace information.
`FullDOM`	Sample usage of full DOM interface. Exercises all the calls.

Table 14–6 (Cont.) XML Parser for C: Sample Built Programs

Built Program	Description
`XSLSample` *xmlfile xslss*	Sample usage of XSL processor. It takes two file names as input, the XML file and XSL stylesheet
`XVMSample` *xmlfile xslfile*	Sample usage of the XSLT Virtual Machine and Compiler. It takes two files as input - the XML file and the XSL stylesheet.
`XSLXPathSample` *xmlfile xpathexpr*	Sample usage of XSL/XPath processor. It takes an XML file and an XPath expression as input. Generates the result of the evaluated XPath expression.
`XVMXPathSample` *xmlfile xpathexpr*	Sample usage of the XSLT Virtual Machine and Compiler. It takes an XML file and an XPath expression as input. Generates the result of the evaluated XPath expression.

15 XSLT Processors for C

This chapter contains these topics:

- XVM Processor
- XSL 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 14, "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 XSL 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.

```
XmlXvmDestroy(vm);
XmlXvmDestroyComp(comp);
XmlDestroy(xctx);
```

Command-Line Access of the XVM Processor

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 provided with the database and the Application Server. It is also available for download from the OTN site:

See Also:

- *Oracle XML API Reference* "XSLTVM APIs for C"
- http://otn.oracle.com/tech/xml/

XSL 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 XSL Processor and XPath Processor. The Oracle implementation of XSL processor follows the more common design approach, which melts 'compiler' and 'processor' into one object.

XSL 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 XSLT stylesheet.

   ```
   xslDomdoc = XmlLoadDom(xctx, &err, "file", xslFile, "base_uri", baseuri, NULL);
   ```

3. Create an XSL Processor for the stylesheet

```
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);
```

Command Line Usage of the XSL Processor

The Oracle XSL processor for C can be called 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 15–1 lists the command line options.

Table 15–1 XSLT Processor for C: Command Line Options

Option	Description
`-B` *`BaseUri`*	Set the Base URI for XSLT processor: `BaseUri` of `http://pqr/xsl.txt` resolves `pqr.txt` to `http://pqr/pqr.txt`
`-e` *`encoding`*	Specify default input file encoding (`-ee` to force).
`-E` *`encoding`*	Specify DOM or SAX encoding.
`-f`	File - interpret as filespec, not URI.
`-G` *`xptrexprs`*	Evaluates XPointer schema examples given in a file.
`-h`	Help - show this usage. (Use `-hh` for more options.)
`-hh`	Show complete options list.
`-i` *`n`*	Number of times to iterate the XSLT processing.
`-l` *`language`*	Language for error reporting.
`-o` *`XSLoutfile`*	Specifies output file of XSLT processor.
`-v`	Version - display parser version then exit.
`-V` *`var value`*	Test top-level variables in C XSLT.
`-w`	Whitespace - preserve all whitespace.
`-W`	Warning - stop parsing after a warning.

Accessing Oracle XSL Processor for C

Oracle XSL Processor for C is provided with the database and the Application Server. It is also available for download from the OTN site:

See Also:

- *Oracle XML API Reference* "XSLT APIs for C"
- *Oracle XML API Reference* "XPath APIs for C"
- `http://otn.oracle.com/tech/xml/`

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 15–2 lists the files in that directory:

Table 15–2 XSLT for C Demo Files

Sample File Name	Description
`XSLSample.c`	Source for XSLSample program
`XSLSample.std`	Expected output from XSLSample
`class.xml`	XML file that can be used with XSLSample
`iden.xsl`	Stylesheet that can be used with XSLSample
`cleo.xml`	XML version of Shakespeare's play

Table 15–2 (Cont.) XSLT for C Demo Files

Sample File Name	Description
`XVMSample.c`	Sample usage of XSLT Virtual Machine and compiler. It takes two filenames as input - XML file and XSL stylesheet file.
`XVMXPathSample.c`	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.
`XSLXPathSample.c`	Sample usage of `XSL/XPath` processor. It takes XML file name and `XPath` expression as input. Generates the result of the evaluated `XPath` expression.

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 XSL stylesheet:

```
XSLSample xmlfile xslss
```

16

XML Schema Processor for C

This chapter contains these topics:

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 14, "XML Parser for C".

Oracle XML Schema Processor for C

The XML Schema Processor for C is a companion component to the XML Parser for C. It 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.

See Also: Chapter 3, "XML Parser 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

See Also:

- *Oracle XML API Reference* "Schema APIs for C"
- `/xdk/demo/c/schema/` - sample code

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 16–1 lists the supplied files and directories for this release:

Table 16–1 XML Schema Processor for C: Supplied Files

Directory and Files	Description
`bin`	schema processor executable, `schema`
`lib`	XML/XSL/Schema & support libraries
`nls/data`	Globalization Support data files
`xdk/demo/c/schema`	example usage of the Schema processor
`xdk/include`	header files
`xdk/mesg`	error message files
`xdk/readme.html`	introductory file

Table 16–2 lists the included libraries:

Table 16–2 XML Schema Processor for C: Supplied Libraries

Included Library	Description
`libxml10.a`	XML Parser, XSL Processor, XML Schema Processor
`libcore10.a`	CORE functions
`libnls10.a`	Globalization Support

Invoking XML Schema Processor for C

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.

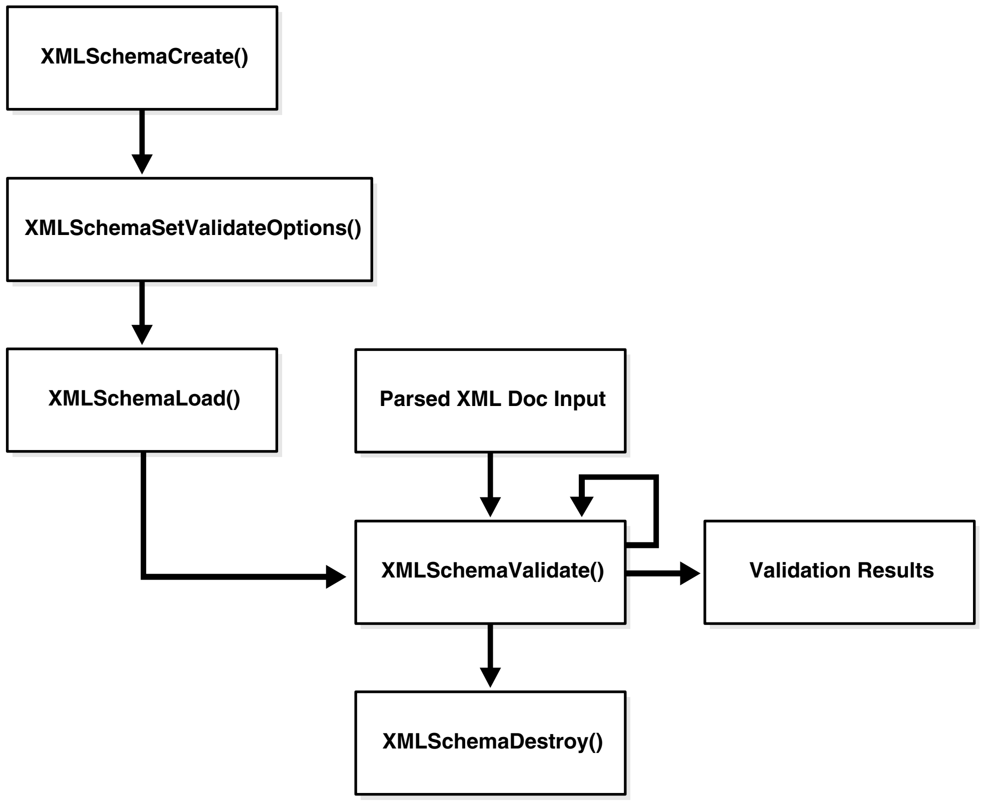
XML Schema Processor for C Usage Diagram

Figure 16–1 describes the calling sequence for the XML Schema Processor for C, as follows:

The sequence of calls to the processor is: initialize, load, validate, validate, ..., validate, terminate.

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.

Figure 16–1 XML Schema Processor for C Usage Diagram



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 16–3 lists the provided sample files.

Table 16–3 XML Schema for C Samples Provided

Sample File	Description
`Makefile`	Makefile to build the sample programs and run them, verifying correct output.
`xsdtest.c`	Program which invokes the XML Schema for C API
`car.{xsd,xml,std}`	Sample schema, instance document, and expected output respectively, after running `xsdtest` on them.
`aq.{xsd,xml,std}`	Second sample schema, instance document, and expected output respectively, after running `xsdtest` on them.
`pub.{xsd,xml,std}`	Third sample schema, instance document, and expected output respectively, after running `xsdtest` on them.

To build the sample programs, run `make`.

To build the programs and run them, comparing the actual output to expected output:

```
make sure
```

17 Getting Started with XDK C++ Components

This chapter contains this topic:

- Installation of the XDK C++ Components

Installation of the XDK C++ Components

XDK C++ components are the basic building blocks for reading, manipulating, and transforming XML documents.

Oracle XDK C++ components consist of the following:

- XML Parser for C++: supports parsing XML documents with the DOM or SAX interfaces.
- XSL Processor for C++: supports transforming XML documents.
- XML Schema Processor for C++: supports parsing and validating XML files against an XML Schema definition file (default extension `.xsd`).
- Class Generator for C++: generates a set of C++ source files based on an input DTD or XML Schema.

Getting the C++ Components of XDK

If you have installed the Oracle Database or Oracle Application Server, you will already have the XDK C++ components installed. You can also download the latest versions of XDK C++ components from OTN.

In order to download the XDK from OTN, follow these steps:

1. Go to the URL `http://www.oracle.com/technology/tech/xml/`
2. Click the XML Developer's Kit link.
3. Logon with your OTN username and password (registration is free if you don't already have an account).
4. Select the version that you want to download.
5. Accept all conditions in the licensing agreement.
6. Click the appropriate `*.tar.gz` or `*.zip` file.
7. Extract the files in the distribution:
   a. Choose a directory under which you would like the `xdk` directory and subdirectories to go.

b. Change to that directory; then extract the XDK download archive file with the tool:

```
UNIX: tar xvfz xdk_xxx.tar.gz
Windows: use WinZip visual archive extraction tool
```

Libraries in the UNIX Environment for C++ XDK

After installing the XDK, the directory structure is:

```
-$XDK_HOME
     | - bin: executable files
     | - lib: library files.
     | - nls/data: Globalization Support data files(*.nlb)
     | - xdk
          | - demo/cpp: demonstration code
          | - doc/cpp: documentation
          | - public: header files
          | - mesg: message files (*.msb)
```

The libraries that come with the UNIX version of XDK C++ components are listed in the following table:

Table 17–1 XDK Libraries for C++ (UNIX)

Component	Library	Notes
XML Parser, XSL Processor, XML Schema Processor, Class Generator	`libxml10.a`	XML Parser V2 for C++, which includes DOM, SAX, and XSLT APIs, XML Schema Processor for C++, Class Generator for C++

The XDK C++ components package depends on the Oracle CORE and Globalization Support libraries, which are listed in the following table:

Table 17–2 Dependent Libraries of XDK C++ Components on UNIX

Component	Library	Notes
CORE Library	`libcore10.a`	Oracle CORE library
Globalization Support Library	`libnls10.a`	Oracle Globalization Support common library
Globalization Support Library	`libunls10.a`	Oracle Globalization Support library for Unicode support

Setting the UNIX Environment for C++

Check that the environment variable `ORA_NLS10` is set to point to the location of the Globalization Support data files. If you install the Oracle database, you can set it to be:

```
setenv ORA_NLS10 ${ORACLE_HOME}/nls/data
```

If no Oracle database is installed, you must use the Globalization Support data files that come with the XDK release:

```
setenv ORA_NLS10 ${XDK_HOME}/nls/data
```

Check that the environment variable `ORA_XML_MESG` is set to point to the absolute path of the `mesg` directory. If you install the Oracle database, although this is not required, you can set it to:

```
setenv ORA_XML_MESG ${ORACLE_HOME}/xdk/mesg
```

If no Oracle database is installed, you must set it to be the directory of the error message files that comes with the XDK release:

```
setenv ORA_XML_MESG ${XDK_HOME}/xdk/mesg
```

The XDK components can be invoked by writing your own programs to use the supplied API. Compile the programs using the headers in the `xdk/include/` subdirectory and link against the libraries in the `lib/` subdirectory. See `Makefile` in the `xdk/demo/` subdirectory for full details of how to build your programs.

Command Line Environment Setup

The parser may be called as an executable by invoking `bin/xml`, which has the following options:

> **See Also:** For information about Command Line Parser usage, see Table 13–3, " Parser Command Line Options"

To get the XDK version you are using on UNIX:

```
strings libxml10.a | grep -i Version
```

You can now use the `Makefiles` to compile and link the demo code and start developing your program using XDK C++ components.

Windows Environment Setup for C++ XDK

These are the Windows libraries that come with the XDK C++ components:

Table 17–3 XDK C++ Components Libraries on Windows

Component	Library	Notes
XML Parser XSL Processor XML Schema Processor Class Generator	`oraxml10.lib` `oraxml10.dll`	XML Parser V2 for C++, which includes DOM, SAX, and XSLT APIs XML Schema Processor for C++ Class Generator for C++

The XDK C++ components (Windows) depends on the Oracle CORE and Globalization Support libraries in the following table:

Table 17–4 Dependent Libraries of XDK C++ Components on Windows

Component	Library	Notes
CORE Library	`oracore10.dll`	Oracle CORE library
Globalization Support Library	`oranls.dll`	Oracle Globalization Support common library
Globalization Support Library	`oraunls.dll`	Oracle Globalization Support library for Unicode support

Environment for Command Line Usage on Windows

Check that the environment variable `ORA_NLS10` is set to point to the location of the Globalization Support data files. If you install the Oracle database:

```
set ORA_NLS10 = %ORACLE_HOME%\nls\data
```

If no Oracle database is installed, you must use the Globalization Support data files that come with the XDK release:

```
set ORA_NLS10 =%XDK_HOME%\nls\data
```

Check that the environment variable `ORA_XML_MESG` is set to point to the absolute path of the `mesg` directory. If you install the Oracle database, although it is not required, you can set it to:

```
set ORA_XML_MESG =%ORACLE_HOME%\xdk\mesg
```

If no Oracle database is installed, you must set it to be the directory of the error message files, which comes with the XDK release:

```
set ORA_XML_MESG =%XDK_HOME%\xdk\mesg
```

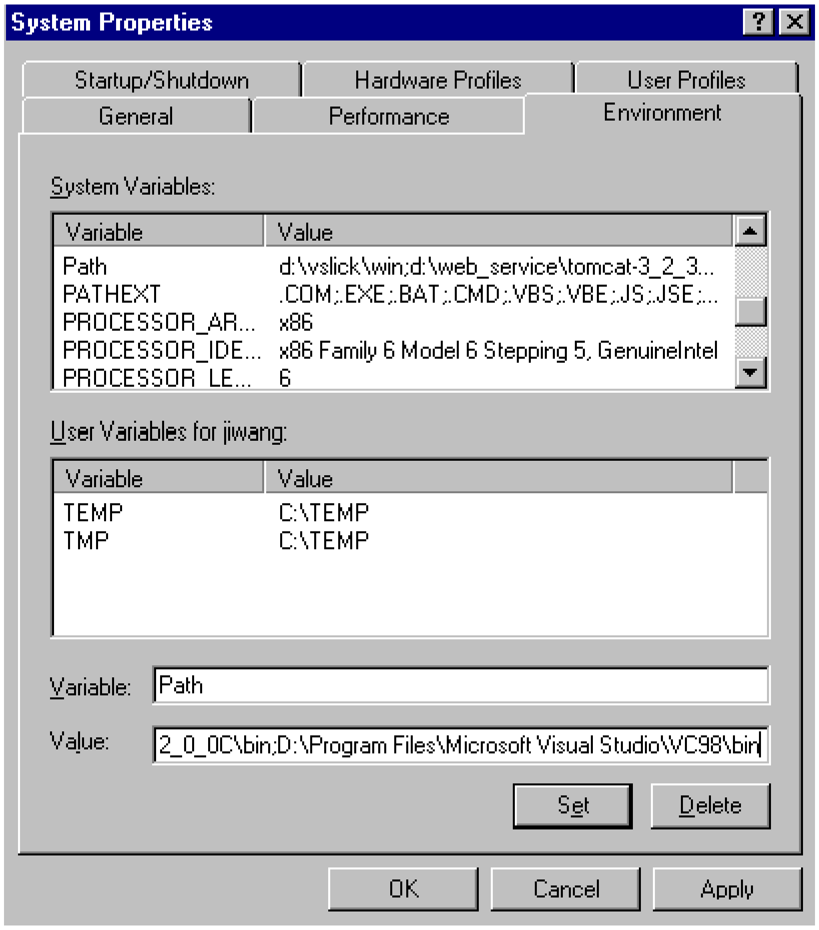
See Also: For information about Command Line Parser usage, see Table 13–3, " Parser Command Line Options"

Setting the Windows Environment for C++ XDK

Set the path for `cl` compiler, if you need to compile the code using `make.bat` in a command line.

Go to the Start Menu and select Settings > Control Panel. In the pop up window of Control Panel, select System icon and double click. A window named System Properties will pop up. Select Environment Tab and input the path of `cl.exe` to the PATH variable shown in Figure 17–1.

Figure 17–1 Setting the PATH for the cl Compiler



System Variables:

Variable	Value
Path	d:\vslick\win;d:\web_service\tomcat-3_2_3...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;...
PROCESSOR_AR...	x86
PROCESSOR_IDE...	x86 Family 6 Model 6 Stepping 5, GenuineIntel
PROCESSOR_LE...	6

User Variables for jiwang:

Variable	Value
TEMP	C:\TEMP
TMP	C:\TEMP

Variable: Path

Value: 2_0_0C\bin;D:\Program Files\Microsoft Visual Studio\VC98\bin

You must update the file `Make.bat` by adding the path of the libraries and header files to the compile and link commands:

```
: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 -
ID:\Progra~1\Micros~1\VC98\Include %filename%.c
goto :EOF

:LINK
set filename=%1
link %link_dbg% /out:..\..\..\..\bin\%filename%.exe /libpath:%ORACLE_HOME%\lib
/libpath:D:\Progra~1\Micros~1\VC98\lib /libpath:..\..\..\..\lib %filename%.obj
oraxml10.lib oracore10.lib oranls10.lib oraunls10.lib user32.lib kernel32.lib
msvcrt.lib ADVAPI32.lib oldnames.lib winmm.lib
:EOF
```

where

`D:\Progra~1\Micros~1\VC98\Include:` is the path for header files and `D:\Progra~1\Micros~1\VC98\lib:` is the path for library files.

Now you can start developing with XDK C++ components.

Using XDK C++ Components with Visual C++

Check that the environment variable `ORA_NLS10` is set to point to the location of the Globalization Support data files.

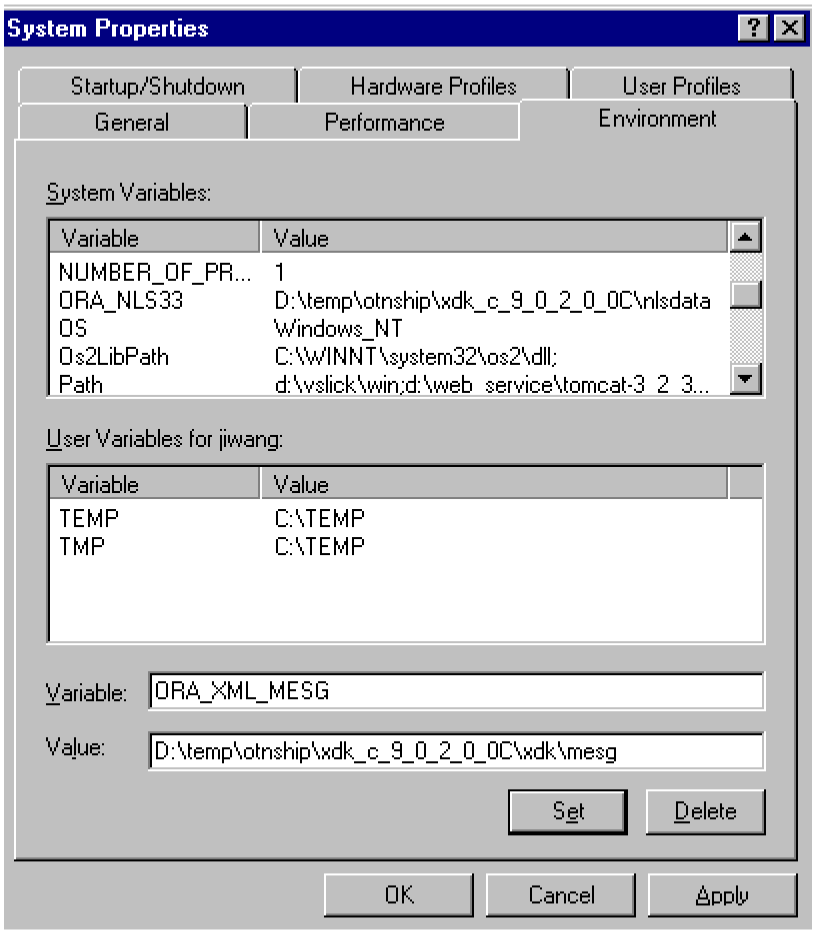
In order for Visual C++ to know the environment variable, you need to use the system setup for Windows to define the environment variable.

Go to Start Menu and select Settings > Control Panel. In the pop-up window of Control Panel, select System icon and double click. A window named System Properties will be popped up. Select Environment Tab and input `ORA_NLS10`.

In order for Visual C++ to employ the environment variable, you need to use the system setup for Windows to define the environment variable.

Go to the Start Menu and select Settings > Control Panel. In the pop-up window of Control Panel, select System icon and double click. A window named System Properties will pop up. Select Environment Tab and input the ORA_XML_MESG, as shown in Figure 17–2:

Figure 17–2 Setting Up ORA_XML_MESG Environment Variable



System Variables:

Variable	Value
NUMBER_OF_PR...	1
ORA_NLS33	D:\temp\otnship\xdk_c_9_0_2_0_0C\nlsdata
OS	Windows_NT
Os2LibPath	C:\WINNT\system32\os2\dll;
Path	d:\vslick\win;d:\web_service\tomcat-3_2_3...

User Variables for jiwang:

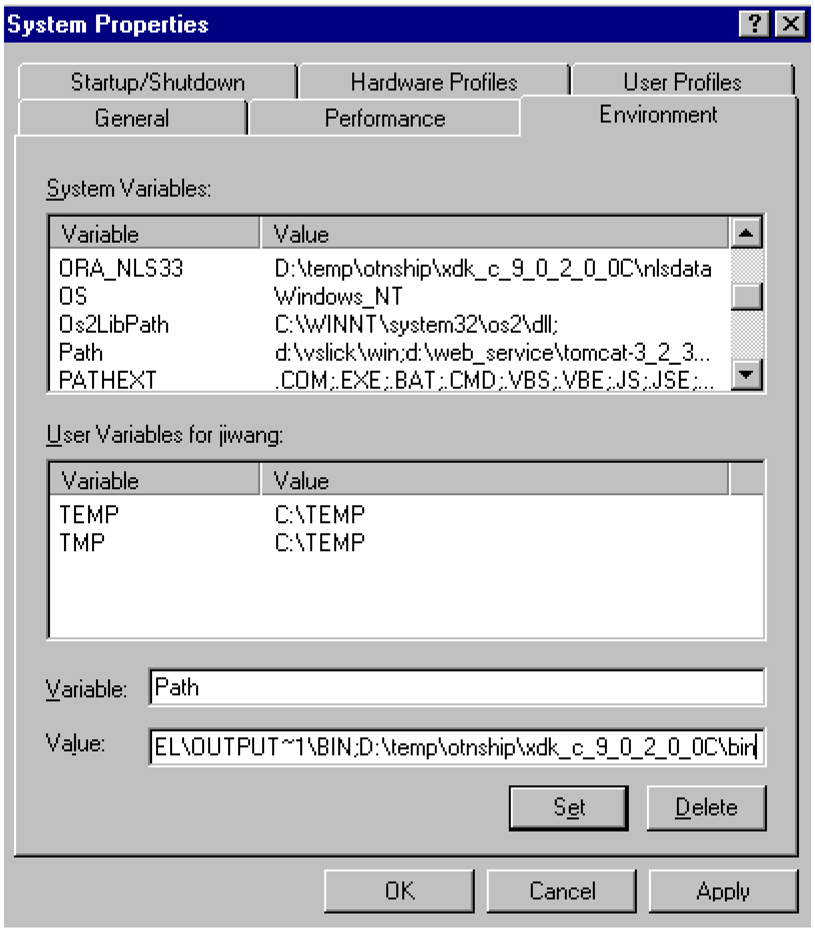
Variable	Value
TEMP	C:\TEMP
TMP	C:\TEMP

Variable: ORA_XML_MESG

Value: D:\temp\otnship\xdk_c_9_0_2_0_0C\xdk\mesg

Figure 17–3, "Setup of the PATH for DLLs" shows how to set up the PATH for DLL libraries:

Figure 17–3 Setup of the PATH for DLLs


System Properties

System Variables:

Variable	Value
ORA_NLS33	D:\temp\otnship\xdk_c_9_0_2_0_0C\nlsdata
OS	Windows_NT
Os2LibPath	C:\WINNT\system32\os2\dll;
Path	d:\vslick\win;d:\web_service\tomcat-3_2_3...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;...

User Variables for jiwang:

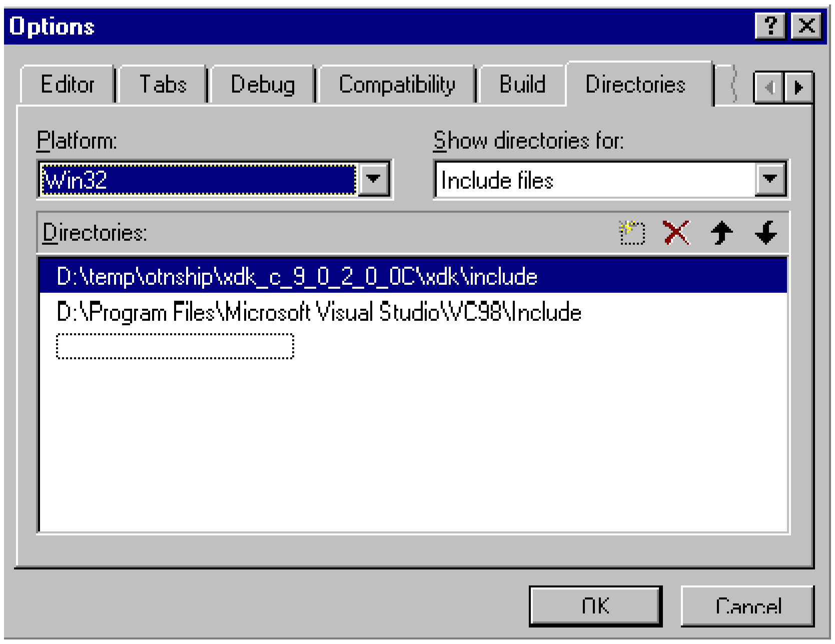
Variable	Value
TEMP	C:\TEMP
TMP	C:\TEMP

Variable: Path

Value: EL\OUTPUT~1\BIN;D:\temp\otnship\xdk_c_9_0_2_0_0C\bin

After you open a workspace in Visual C++ and include the `*.cpp` files for your project, you must set the path for the project. Go to the Tools menu and select Options. A window will pop up. Select the Directory tab and set your include path as shown in Figure 17–4:

Figure 17–4 Setting Your Include Path in Visual C++



Platform: Win32

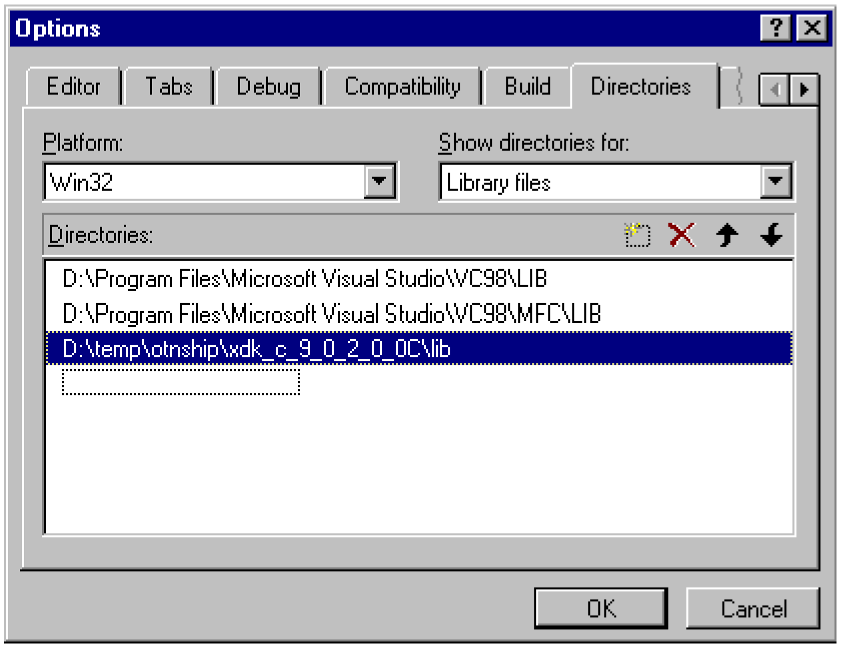
Show directories for: Include files

Directories:

D:\temp\otnship\xdk_c_9_0_2_0_0C\xdk\include
D:\Program Files\Microsoft Visual Studio\VC98\Include

Then set your library path as shown in Figure 17–5:

Figure 17–5 Setting Your Static Library Path in Visual C++



Platform: Win32

Show directories for: Library files

Directories:

D:\Program Files\Microsoft Visual Studio\VC98\LIB
D:\Program Files\Microsoft Visual Studio\VC98\MFC\LIB
D:\temp\otnship\xdk_c_9_0_2_0_0C\lib

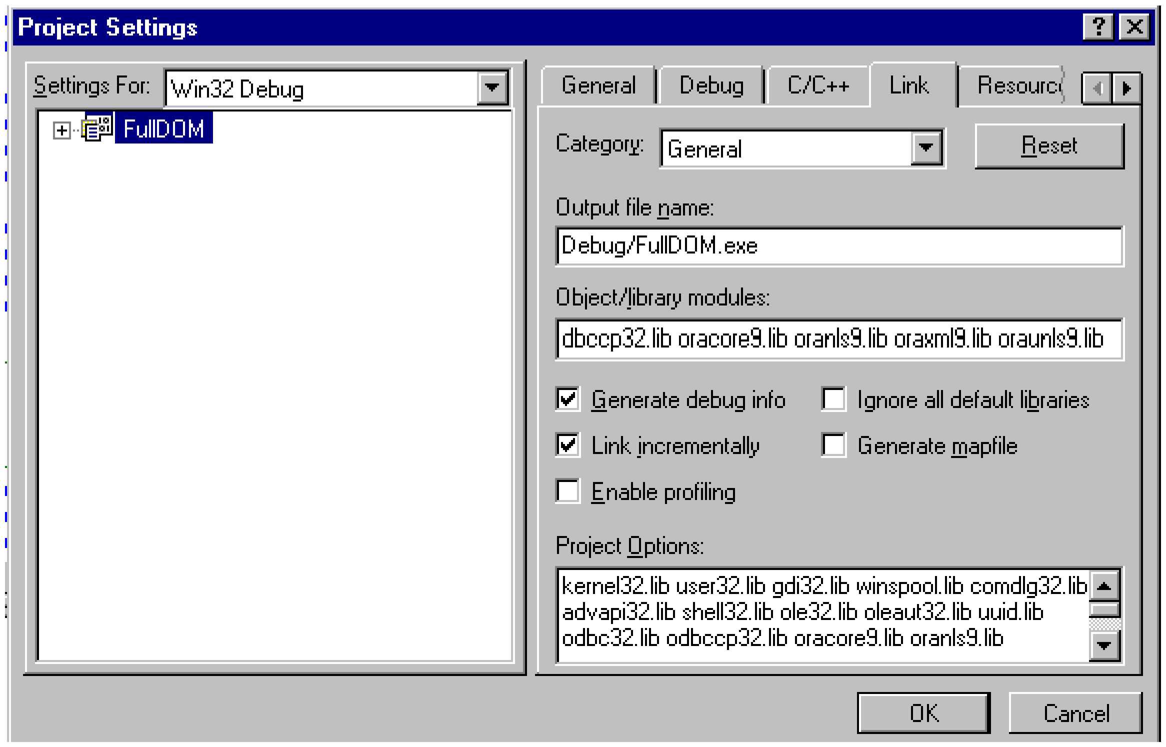
This illustration is of an Options window in Visual C++ as described in the section "Using XDK C++ Components with Visual C++" that surrounds it. The tab "Directories" is selected. The static library path is set in this window.

****

After setting the paths for the static libraries in %XDK_HOME%\lib, you also need to set the library name in the compiling environment of Visual C++.

Go to the Project menu in the menu bar and select Settings. A window will pop up. Please select the Link tab in the Object/Library Modules field enter the name of XDK C++ components libraries, as shown in Figure 17–6:

Figure 17–6 Setting Up the Static Libraries in Visual C++ Project



Settings For: Win32 Debug

FullDOM

Category: General

Output file name: Debug/FullDOM.exe

Object/library modules: dbccp32.lib oracore9.lib oranls9.lib oraxml9.lib oraunls9.lib

[✓] Generate debug info
[ ] Ignore all default libraries
[✓] Link incrementally
[ ] Generate mapfile
[ ] Enable profiling

Project Options:

```
kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib
advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib
odbc32.lib odbccp32.lib oracore9.lib oranls9.lib
```

You can now compile and run the demo programs, and start using XDK C++ components.

> **See Also:** Chapter 19, "XML Parser for C++" for further discussion of the XDK C++ components

18 Unified C++ Interfaces

This chapter contains these topics:

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 the database and the Oracle Application Server and is also available for download from the OTN site: `http://www.oracle.com/technology/tech/xml`.

Sample files are located at `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 shared server 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 XML API Reference*

19 XML Parser for C++

This chapter contains these topics:

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.

Introduction to Parser for C++

Oracle XML Parser for C++ checks if 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`. 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.

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++ will check if an XML document is well-formed, and optionally validate it against a DTD. The parser will construct an object tree which can be accessed through a DOM interface or operate 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 fully conform 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 19–1 lists the sample files in the directory. Each file `*Main.cpp` has a corresponding `*Gen.cpp` and `*Gen.hpp`.

Table 19–1 XML Parser for C++ Sample Files

Sample File Name	Description
`DOMSampleMain.cpp`	Sample usage of C++ interfaces of XML Parser and DOM.
`FullDOMSampleMain.cpp`	Manually build DOM and then exercise.
`SAXSampleMain.cpp`	Source for SAXSample program.

See Also: *Oracle XML API Reference*

20

XSLT Processor for C++

This chapter contains these topics:

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 the database and the Application Server. It is also available for download from the OTN site `http://www.oracle.com/technology/tech/xml`.

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 15-1

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.

See Also: *Oracle XML API Reference*

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 14–4, " XML Parser and XSLT Processor: 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 20–1 lists the sample files.

Table 20–1 XSLT for C++ Sample Files

Sample File Name	Description

`XSLSampleMain.cpp`
`XSLSampleGen.cpp`
`XSLSampleGen.hpp`
	Sources for sample XSLT usage program. `XSLSample` takes two arguments, the XSL 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.

`XVMSampleMain.cpp`
`XVMSampleGen.cpp`
`XVMSampleGen.hpp`
	Sources for the sample XVM usage program.

21

XML Schema Processor for C++

This chapter contains these topics:

- Oracle XML Schema Processor for C++
- XML Schema Processor API
- Running the Provided XML Schema for C++ 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.

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 XML API Reference*

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 21–1 lists the options (can be listed if the option is invalid or -h is the option):

Table 21–1 XML Schema Processor for C++ Command Line Options

Option	Description
`-0`	Always exit with code 0 (success).
`-e` *encoding*	Specify default input file encoding.
`-E` *encoding*	Specify output/data/presentation encoding.
`-h`	Help. Prints these choices.
`-i`	Ignore provided schema.
`-o` *num*	Validation option.
`-p`	Print document instance to `stdout` on success.
`-u`	Force the Unicode path.
`-v`	Version - display version, then exit.

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 21–2 lists the sample files provided.

Table 21–2 XML Schema Processor for C++ Samples Provided

Sample File	Description
`Makefile`	Makefile to build the sample programs and run them, verifying correct output.
`xsdtest.cpp`	Trivial program which invokes the XML Schema for C++ API
`car.{xsd,xml,std}`	Sample schema, instance document, expected output respectively, after running `xsdtest` on them.
`aq.{xsd,xml,std}`	Second sample schema's, instance document, expected output respectively, after running `xsdtest` on them.
`pub.{xsd,xml,std}`	Third sample schema's, instance document, expected output respectively, after running `xsdtest` on them.

To build the sample programs, run `make`.

To build the programs and run them, comparing the actual output to expected output:

```
make sure
```

22 XPath Processor for C++

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;

and XvmXPathSampleGen.hpp, XvmXPathSampleGen.cpp, XvmXPathSampleMain.cpp, XvmXPathSampleForce.cpp.

See Also: *Oracle XML API Reference*

23 XML Class Generator for C++

This chapter contains these topics:

Accessing XML C++ Class Generator

The XML C++ Class Generator is provided with the database and is also available for download from the OTN site `http://www.oracle.com/technology/tech/xml`.

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 not required.

XML C++ Class Generator Usage

The standalone class generator can be called as an executable by invoking `bin/xmlcg`.

1. The C++ command line class generator, `bin/xmlcg` is invoked in the following way:

   ```
   xmlcg [options] input_file
   ```

   where the options are described in Table 23–1:

   Table 23–1 Class Generator Options

Option	Meaning
`-d` *name*	Input is an external DTD or a DTD file. Generates `name.cpp` and `name.h.`
`-o` *directory*	Output directory for generated files (the default is the current directory).
`-e` *encoding*	Default input file encoding.
`-h`	Show this usage help.
-v	Show the class generator version validator options.
-s *name*	Input is an XML Schema file with the given name. Generates `name.cpp` and `name.h`.

   `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 above 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 23–2 lists the demo XML C++ Class Generator files:

Table 23–2 XML C++ Class Generator Files

File Name	Description
`CG.cpp`	Sample program
`CG.xml`	XML file contains DTD and dummy document
`CG.dtd`	DTD file referenced by CG.xml
`Make.bat` on Windows NT `Makefile` on UNIX	Batch file (on Windows) or Make file (on UNIX) to generate classes and build the sample programs.
README	A readme file with these instructions

The `make.bat` batch file (on Windows NT) 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 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.

```
<!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.
4. Invokes the validation function which verifies that the constructed classes match the DTD.
5. Writes the constructed document to `Sample.xml`.

```
//////////////////////////////////////////////////////////////////////////////
// NAME        CG.cpp
// DESCRIPTION Demonstration program for C++ Class Generator usage
//////////////////////////////////////////////////////////////////////////////

#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 fstream(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
```

24 XSU for PL/SQL

This chapter contains these topics:

> **See Also::** Chapter 7, "XML SQL Utility (XSU)" for information about XSU in general.

XSU PL/SQL API

XML SQL Utility (XSU) PL/SQL API reflects the Java API in the generation and storage of XML documents from and to a database. `DBMS_XMLQuery` and `DBMS_XMLSave` are the two packages that reflect the functions in the Java classes - `OracleXMLQuery` and `OracleXMLSave`. Both of these packages have a context handle associated with them. Create a context by calling one of the constructor-like functions to get the handle and then use the handle in all subsequent calls.

XSU Supports XMLType

From Oracle9i Release 2 (9.2), XSU supports XMLType. Using XSU with XMLType is useful if, for example, you have XMLType columns in objects or tables.

> **See Also:** *Oracle XML DB Developer's Guide,* in particular, the chapter on Generating XML, for examples on using XSU with XMLType.

Generating XML with DBMS_XMLQuery()

Generating XML results in a CLOB that contains the XML document. To use `DBMS_XMLQuery` and the XSU generation engine, follow these steps:

1. Create a context handle by calling the `DBMS_XMLQuery.getCtx` function and supplying it the query, either as a `CLOB` or a `VARCHAR2`.
2. Bind possible values to the query using the `DBMS_XMLQuery.bind` function. The binds work by binding a name to the position. For example, the query can be

select * from employees where employee_id = :EMPNO_VAR. Here you are binding the value for the EMPNO_VAR using the setBindValue function.

3. Set optional arguments like the ROW tag name, the ROWSET tag name, or the number of rows to fetch, and so on.
4. Fetch the XML as a CLOB using the getXML() functions. getXML() can be called to generate the XML with or without a DTD or schema.
5. Close the context.

Here are some examples that use the DBMS_XMLQuery PL/SQL package.

XSU Generating XML Example 1: Generating XML from Simple Queries (PL/SQL)

In this example, you select rows from table employees, and obtain an XML document as a CLOB. First get the context handle by passing in a query and then call the getXMLClob routine to get the CLOB value. The document is in the same encoding as the database character set.

```
declare
  queryCtx DBMS_XMLquery.ctxType;
  result CLOB;
begin

  -- set up the query context...!
  queryCtx := DBMS_XMLQuery.newContext('select * from employees');

  -- get the result..!
  result := DBMS_XMLQuery.getXML(queryCtx);
  -- Now you can use the result to put it in tables/send as messages..
  printClobOut(result);
  DBMS_XMLQuery.closeContext(queryCtx);  -- you must close the query handle..
end;
```

XSU Generating XML Example 2: Printing CLOB to Output Buffer

printClobOut() is a simple function that prints the CLOB to the output buffer. If you run this PL/SQL code in SQL*Plus, the result of the CLOB is printed to the screen. Set the serveroutput to on in order to see the results. You may have to increase your display buffer to see all the output:

```
set serveroutput on size 200000
set long 20000
```

Here is the code:

```
CCREATE OR REPLACE PROCEDURE printClobOut(result IN OUT NOCOPY CLOB) is
xmlstr varchar2(32767);
line varchar2(2000);
begin
  xmlstr := dbms_lob.SUBSTR(result,32767);
  loop
    exit when xmlstr is null;
    line := substr(xmlstr,1,instr(xmlstr,chr(10))-1);
    dbms_output.put_line('| '||line);
    xmlstr := substr(xmlstr,instr(xmlstr,chr(10))+1);
  end loop;
end;
```

XSU Generating XML Example 3: Changing ROW and ROWSET Tag Names

With the XSU PL/SQL API you can also change the `ROW` and the `ROWSET` tag names. These are the default names placed around each row of the result, and round the whole document, respectively. The procedures, `setRowTagName` and `setRowSetTagName` accomplish this as shown in the following example:

```
--Setting the ROW tag names

declare
   queryCtx DBMS_XMLQuery.ctxType;
   result CLOB;
begin
   -- set the query context.
   queryCtx := DBMS_XMLQuery.newContext('select * from employees');

   DBMS_XMLQuery.setRowTag(queryCtx,'EMP');       -- sets the row tag name
   DBMS_XMLQuery.setRowSetTag(queryCtx,'EMPSET'); -- sets rowset tag name

   result := DBMS_XMLQuery.getXML(queryCtx);      -- get the result

   printClobOut(result);  -- print the result..!
   DBMS_XMLQuery.closeContext(queryCtx);          -- close the query handle;
end;
```

The resulting XML document has an `EMPSET` document element. Each row is separated using the `EMP` tag.

XSU Generating XML Example 4: Using setMaxRows() and setSkipRows()

The results from the query generation can be paginated by using:

- `setMaxRows` function. This sets the maximum number of rows to be converted to XML. This is relative to the current row position from which the last result was generated.
- `setSkipRows` function. This specifies the number of rows to skip before converting the row values to XML.

For example, to skip the first 3 rows of the employees table and then print out the rest of the rows 10 at a time, you can set the `skipRows` to 3 for the first batch of 10 rows and then set `skipRows` to 0 for the rest of the batches.

As in the case of XML SQL Utility's Java API, call the `keepObjectOpen()` function to ensure that the state is maintained between fetches. The default behavior is to close the state after a fetch. For multiple fetches, you must determine when there are no more rows to fetch. This can be done by setting the `setRaiseNoRowsException()`. This causes an exception to be raised if no rows are written to the CLOB. This can be caught and used as the termination condition.

```
-- Pagination of results

--Setting the ROW tag names

declare
   queryCtx DBMS_XMLQuery.ctxType;
   result CLOB;
begin
   -- set the query context.
   queryCtx := DBMS_XMLQuery.newContext('select * from employees');
   DBMS_XMLQuery.setRowTag(queryCtx,'EMP'); -- sets the row tag name
   DBMS_XMLQuery.setRowSetTag(queryCtx,'EMPSET'); -- sets rowset tag name

   result := DBMS_XMLQuery.getXML(queryCtx); -- get the result

   printClobOut(result);  -- print the result..!
   DBMS_XMLQuery.closeContext(queryCtx);  -- close the query handle;
end;
```

Setting Stylesheets in XSU (PL/SQL)

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 XML. To do this, use `setStylesheetHeader()` procedure, to set the stylesheet header in the result. This simply adds the XML processing instruction to include the stylesheet.
- Apply a stylesheet to the result XML document, before generation. This method is a huge performance win since otherwise the XML document has to be generated as a CLOB, sent to the parser again, and then 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. This uses the stylesheet to generate the result.

Binding Values in XSU (PL/SQL)

The XSU PL/SQL API provides the ability to bind values to the SQL statement. The SQL statement can contain named bind variables. The variables must be prefixed with a colon (:) to declare that they are bind variables. To use the bind variable follow these steps:

1. ***Initialize the query context with the query containing the bind variables.*** For example, the following statement registers a query to select the rows from the `employees` table with the where clause containing the bind variables `:EMPLOYEE_ID` and `:FIRST_NAME`. You will bind the values for employee number and employee first name later.

   ```
   queryCtx = DBMS_XMLQuery.getCtx('select * from employees where employee_id =
              :EMPLOYEE_ID and first_name = :FIRST_NAME');
   ```

2. ***Set the list of bind values.*** The `clearBindValues()` clears all the bind variables set. The `setBindValue()` sets a single bind variable with a string value. For example, you will set the `empno` and `ename` values as shown later:

   ```
   DBMS_XMLQuery.clearBindValues(queryCtx);
   DBMS_XMLQuery.setBindValue(queryCtx,'EMPLOYEE_ID',20);
   DBMS_XMLQuery.setBindValue(queryCtx,'FIRST_NAME','John');
   ```

3. ***Fetch the results.*** This will apply the bind values to the statement and then get the result corresponding to the predicate `employee_id = 20` and `first_name = 'John'`.

   ```
   DBMS_XMLQuery.getXMLClob(queryCtx);
   ```

4. ***Re-bind values if necessary***. For example to change the `FIRST_NAME` alone to `scott` and reexecute the query,

   ```
   DBMS_XMLQuery.setBindValue(queryCtx,'FIRST_NAME','Scott');
   ```

The rebinding of `FIRST_NAME` will now use `Scott` instead of `John`.

XSU Generating XML Example 5: Binding Values to the SQL Statement

The following example illustrates the use of bind variables in the SQL statement:

```
declare
  queryCtx DBMS_XMLquery.ctxType;
  result CLOB;
begin

queryCtx := DBMS_XMLQuery.newContext('select * from employees where employee_id
         = :EMPLOYEE_ID and first_name = :FIRST_NAME');

--No longer needed:
--DBMS_XMLQuery.clearBindValues(queryCtx);
DBMS_XMLQuery.setBindValue(queryCtx,'EMPLOYEE_ID',100);
DBMS_XMLQuery.setBindValue(queryCtx,'FIRST_NAME','Steven');

result := DBMS_XMLQuery.getXML(queryCtx);

--printClobOut(result);

DBMS_XMLQuery.setBindValue(queryCtx,'FIRST_NAME','Neena');

result := DBMS_XMLQuery.getXML(queryCtx);

--printClobOut(result);
end;

create or replace procedure insProc(xmlDoc IN CLOB, tableName IN VARCHAR2) is
   insCtx DBMS_XMLSave.ctxType;
   rows number;
 begin
    insCtx := DBMS_XMLSave.newContext(tableName);  -- get the context handle
    rows := DBMS_XMLSave.insertXML(insCtx,xmlDoc); -- this inserts the document
    DBMS_XMLSave.closeContext(insCtx);             -- this closes the handle
end;
```

Storing XML in the Database Using DBMS_XMLSave

To use `DBMS_XMLSave` and XML SQL Utility storage engine, follow these steps:

1. Create a context handle by calling the `DBMS_XMLSave.getCtx` function and supplying it the table name to use for the DML operations.
2. ***For inserts.*** You can set the list of columns to insert into using the `setUpdateColNames` function. The default is to insert values into all the columns.

   For updates. The list of key columns must be supplied. Optionally the list of columns to update may also be supplied. 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 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 you can set the list of key columns. In this case only those tag values whose tag names match these columns will be used to identify the rows to delete (in effect used in the WHERE clause of the `DELETE` statement).

3. Supply an XML document to the `insertXML`, `updateXML`, or `deleteXML` functions to insert, update and delete respectively.
4. You can repeat the last operation any number of times.
5. Close the context.

Use the same examples as for the Java case, `OracleXMLSave` class examples.

Insert Processing Using XSU (PL/SQL API)

To insert a document into a table or view, simply 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 an absent element is treated as a NULL value.

The following code shows how the document generated from the `employees` table can be put back into it with relative ease.

XSU Inserting XML Example 6: Inserting Values into All Columns (PL/SQL)

This example creates a procedure, `insProc`, which takes in:

- An XML document as a CLOB
- A table name to put the document into

and then inserts the XML document into the table:

```
create or replace procedure insProc(xmlDoc IN CLOB, tableName IN VARCHAR2) is
   insCtx DBMS_XMLSave.ctxType;
   rows number;
 begin
    insCtx := DBMS_XMLSave.newContext(tableName); -- get the context handle
    rows := DBMS_XMLSave.insertXML(insCtx,xmlDoc); -- this inserts the document
    DBMS_XMLSave.closeContext(insCtx);            -- this closes the handle
end;
```

This procedure can now be called with any XML document and a table name. For example, a call of the form:

```
execute insProc(xmlDocument, 'hr.employees');
```

generates an INSERT statement of the form:

```
INSERT INTO hr.employees (employee_id, last_name, job_id, manager_id, hire_date,
       salary, department_id VALUES(?,?,?,?,?,?,?);
```

and the element tags in the input XML document matching the column names will be matched and their values bound. For the code snippet shown earlier, if you send it the following XML document:

```
<?xml version='1.0'?>
<ROWSET>
  <ROW num="1">
    <EMPLOYEE_ID>7369</EMPLOYEE_ID>
    <LAST_NAME>Smith</LAST_NAME>
    <JOB_ID>CLERK</JOB_ID>
    <MANAGER_ID>7902</MANAGER_ID>
    <HIRE_DATE>12/17/1980 0:0:0</HIRE_DATE>
    <SALARY>800</SALARY>_
    <DEPARTMENT_ID>20</DEPARTMENT_ID>
  </ROW>
  <!-- additional rows ... -->
</ROWSET>
```

you would have a new row in the `employees` table containing the values 7369, Smith, CLERK, 7902, 12/17/1980, 800, 20 for the columns named. Any element absent inside the row element would is considered a null value.

XSU Inserting XML Example 7: Inserting Values into Certain Columns (PL/SQL)

In certain cases, you may not want to insert values into all columns. This might be true when the values that you are getting is not the complete set and you need triggers or default values to be used for the rest of the columns. The code that appears later shows how this can be done.

Assume that you are getting the values only for the employee number, name, and job, and that the salary, manager, department number and hiredate fields are filled in automatically. You create a list of column names that you want the insert to work on and then pass it to the `DBMS_XMLSave` procedure. The setting of these values can be done by calling `setUpdateColumnName()` procedure repeatedly, passing in a column name to update every time. The column name settings can be cleared using `clearUpdateColumnNames()`.

```
create or replace procedure testInsert( xmlDoc IN clob) is
  insCtx DBMS_XMLSave.ctxType;
  doc clob;
  rows number;
begin

   insCtx := DBMS_XMLSave.newContext('hr.employees'); -- get the save context..!

   DBMS_XMLSave.clearUpdateColumnList(insCtx); -- clear the update settings

   -- set the columns to be updated as a list of values..
   DBMS_XMLSave.setUpdateColumn(insCtx,'EMPLOYEE_ID');
   DBMS_XMLSave.setUpdateColumn(insCtx,'LAST_NAME');
   DBMS_XMLSave.setUpdatecolumn(insCtx,'JOB_ID');

   -- Now insert the doc. This will only insert into EMPLOYEE_ID, LAST_NAME, and
   --   JOB_ID columns
   rows := DBMS_XMLSave.insertXML(insCtx, xmlDoc);
   DBMS_XMLSave.closeContext(insCtx);

end;
/
```

If you call the procedure passing in a CLOB as a document, an INSERT statement of the form:

```
INSERT INTO hr.employees (employee_id, last_name, job_id) VALUES (?, ?, ?);
```

is generated. Note that in the earlier example, if the inserted document contains values for the other columns (`HIRE_DATE`, and so on), those are ignored.

An insert is performed for each `ROW` element that is present in the input. These inserts are batched by default.

Update Processing Using XSU (PL/SQL API)

Now that you know how to insert values into the table from XML documents, let us see how to update only certain values. If you get an XML document to update the salary of an employee and also the department that she works in:

```
<ROWSET>
  <ROW num="1">
    <EMPLOYEE_ID>7369</EMPLOYEE_ID>
    <SALARY>1800</SALARY>
    <DEPARTMENT_ID>30</DEPARTMENT_ID>
  </ROW>
  <ROW>
    <EMPLOYEE_ID>2290</EMPLOYEE_ID>
    <SAARY>2000</SALARY>
    <HIRE_DATE>12/31/1992</HIRE_DATE>
  <!-- additional rows ... -->
</ROWSET>
```

you can call the update processing to update the values. In the case of update, you need to supply XSU with the list of key column names. These form part of the WHERE clause in the UPDATE statement. In the employees table shown earlier, the employee number (EMPLOYEE_ID) column forms the key and you use that for updates.

XSU Updating XML Example 8: Updating XML Document Key Columns (PL/SQL)

Consider the PL/SQL procedure:

```
create or replace procedure testUpdate ( xmlDoc IN clob) is
  updCtx DBMS_XMLSave.ctxType;
  rows number;
begin

   updCtx := DBMS_XMLSave.newContext('hr.employees');  -- get the context
   DBMS_XMLSave.clearUpdateColumnList(updCtx);   -- clear the update settings..

   DBMS_XMLSave.setKeyColumn(updCtx,'EMPLOYEE_ID'); -- set EMPLOYEE_ID as key
column
   rows := DBMS_XMLSave.updateXML(updCtx,xmlDoc);  -- update the table.
   DBMS_XMLSave.closeContext(updCtx);              -- close the context..!

end;
/
```

In this example, when the procedure is executed with a CLOB value that contains the document described earlier, two UPDATE statements would be generated. For the first ROW element, you would generate an UPDATE statement to update the fields as shown next:

```
UPDATE hr.employees SET salary = 1800 AND department_id = 30 WHERE employee_id = 7369;
```

and for the second ROW element,

```
UPDATE hr.employees SET salary = 2000 AND hire_date = 12/31/1992 WHERE employee_id = 2290;
```

XSU Updating XML Example 9: Specifying a List of Columns to Update (PL/SQL)

You may want to specify the list of columns to update. This would speed up the processing since the same UPDATE statement can be used for all the ROW elements. Also you can ignore other tags which occur in the document. Note that when you specify a list of columns to update, an element corresponding to one of the update columns, if absent, will be treated as NULL.

If you know that all the elements to be updated are the same for all the ROW elements in the XML document, then you can use the setUpdateColumnName*()* procedure to set the column name to update.

```
create or replace procedure testUpdate(xmlDoc IN CLOB) is
  updCtx DBMS_XMLSave.ctxType;
  rows number;
begin

   updCtx := DBMS_XMLSave.newContext('hr.employees');
   DBMS_XMLSave.setKeyColumn(updCtx,'EMPLOYEE_ID'); -- set EMPLOYEE_ID as key
column

   -- set list of columnst to update.
   DBMS_XMLSave.setUpdateColumn(updCtx,'SALARY');
   DBMS_XMLSave.setUpdateColumn(updCtx,'JOB_ID');

   rows := DBMS_XMLSave.updateXML(updCtx,xmlDoc); -- update the XML document..!
   DBMS_XMLSave.closeContext(updCtx);   -- close the handle

end;
```

Delete Processing Using XSU (PL/SQL API)

For deletes, you can set the list of key columns. These columns will be put as part of the WHERE clause of the DELETE statement. If the key column names are not supplied, then a new DELETE statement will be created for each ROW element of the XML document where the list of columns in the WHERE clause of the DELETE will match those in the ROW element.

XSU Deleting XML Example 10: Deleting Operations for Each Row (PL/SQL)

Consider the delete example shown here:

```
create or replace procedure testDelete(xmlDoc IN clob) is
  delCtx DBMS_XMLSave.ctxType;
  rows number;
begin

   delCtx  := DBMS_XMLSave.newContext('hr.employees');
   DBMS_XMLSave.setKeyColumn(delCtx,'EMPLOYEE_ID');

   rows := DBMS_XMLSave.deleteXML(delCtx,xmlDoc);
   DBMS_XMLSave.closeContext(delCtx);
end;
```

If you use the same XML document shown for the update example, you would end up with two DELETE statements,

```
DELETE FROM hr.employees WHERE employee_id=7369 AND salary=1800 AND department_id=30;
DELETE FROM hr.employees WHERE employee_id=2200 AND salary=2000 AND hire_date=12/31/1992;
```

The DELETE statements were formed based on the tag names present in each ROW element in the XML document.

XSU Example 11: Deleting by Specifying the Key Values (PL/SQL)

If instead you want the delete to only use the key values as predicates, you can use the `setKeyColumn` function to set this.

```
create or replace package testDML AS
   saveCtx DBMS_XMLSave.ctxType := null;   -- a single static variable

   procedure insertXML(xmlDoc in clob);
   procedure updateXML(xmlDoc in clob);
   procedure deleteXML(xmlDoc in clob);

 end;

create or replace package body testDML AS

  rows number;

  procedure insertXML(xmlDoc in clob) is
  begin
    rows := DBMS_XMLSave.insertXML(saveCtx,xmlDoc);
  end;

  procedure updateXML(xmlDoc in clob) is
  begin
    rows := DBMS_XMLSave.updateXML(saveCtx,xmlDoc);
  end;

  procedure deleteXML(xmlDoc in clob) is
  begin
    rows := DBMS_XMLSave.deleteXML(saveCtx,xmlDoc);
  end;

begin
  saveCtx := DBMS_XMLSave.newContext('hr.employees'); -- create the context once
  DBMS_XMLSave.setKeyColumn(saveCtx, 'EMPLOYEE_ID');  -- set the key column name.
end;
```

Here a single `DELETE` statement of the form,

```
DELETE FROM hr.employees WHERE employee_id=?
```

will be generated and used for all `ROW` elements in the document.

XSU Deleting XML Example 12: Reusing the Context Handle (PL/SQL)

In all the three cases described earlier (insert, update, and delete) the same context handle can be used to do more than one operation. That is, you can perform more than one insert using the same context provided all of those inserts are going to the same table that was specified when creating the `save` context. The context can also be used to mix updates, deletes, and inserts.

For example, the following code shows how one can use the same context and settings to insert, delete, or update values depending on the user's 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 the function calls.

```
create or replace package testDML AS
   saveCtx DBMS_XMLSave.ctxType := null;   -- a single static variable
   procedure insert(xmlDoc in clob);
   procedure update(xmlDoc in clob);
   procedure delete(xmlDoc in clob);

 end;
/

create or replace package body testDML AS

  procedure insert(xmlDoc in clob) is
  row number;
  begin
    row := DBMS_XMLSave.insertXML(saveCtx, xmlDoc);
  end;

  procedure update(xmlDoc in clob) is
  begin
    row := DBMS_XMLSave.updateXML(saveCtx, xmlDoc);
  end;

  procedure delete(xmlDoc in clob) is
  begin
    row := DBMS_XMLSave.deleteXML(saveCtx, xmlDoc);
  end;

  begin
    saveCtx := DBMS_XMLSave.newContext('hr.employees'); -- create the context
once..!
    DBMS_XMLSave.setKeyColumn(saveCtx, 'EMPLOYEE_ID');   -- set the key column
name.
  end;
end;
/
```

In the earlier package, you create a context once for the whole package (thus the session) and then reuse the same context for performing inserts, updates and deletes.

Note: The key column `EMPNO` would be used both for updates and deletes as a way of identifying the row.

Users of this package can now call any of the three routines to update the `employees` table:

```
testDML.insert(xmlclob);
testDML.delete(xmlclob);
testDML.update(xmlclob);
```

All of these calls would use the same context. This would improve the performance of these operations, particularly if these operations are performed frequently.

XSU Exception Handling in PL/SQL

Here is an XSU PL/SQL exception handling example:

```
declare
  queryCtx DBMS_XMLQuery.ctxType;
  result clob;
  errorNum NUMBER;
  errorMsg VARCHAR2(200);
```

```
begin

  queryCtx := DBMS_XMLQuery.newContext('select * from employees where df = dfdf');

  -- set the raise exception to true..
  DBMS_XMLQuery.setRaiseException(queryCtx, true);
  DBMS_XMLQuery.setRaiseNoRowsException(queryCtx, true);

  -- set propagate original exception to true to get the original exception..!
  DBMS_XMLQuery.propagateOriginalException(queryCtx,true);
  result := DBMS_XMLQuery.getXML(queryCtx);

  exception
    when others then
      -- get the original exception
      DBMS_XMLQuery.getExceptionContent(queryCtx,errorNum, errorMsg);
      dbms_output.put_line(' Exception caught ' || TO_CHAR(errorNum)
                   || errorMsg );
end;
/
```

Glossary

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.

API

Application Program Interface. See application program interface.

application programming interface (API)

A set of public programmatic interfaces that consist of a language and message format to communicate with an operating system or other programmatic environment, such as databases, Web servers, JVMs, and so forth. These messages typically call functions and methods available for application development.

application server

A server designed to host applications and their environments, permitting server applications to run. A typical example is OAS, which is able to host Java, C, C++, and PL/SQL applications in cases where a remote client controls the interface. See also Oracle Application Server.

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.

BFILE

External binary files that exist outside the database tablespaces residing in the operating system. `BFILE`s are referenced from the database semantics, and are also known as External LOBs.

binary large object (BLOB)

A large object datatype whose content consists of binary data. Additionally, this data is considered raw because its structure is not recognized by the database.

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 Oracle8 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.

CGI

See Common Gateway Interface.

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.

client/server

The term used to describe the application architecture where the actual application runs on the client but accesses data or other external processes on a server across a network.

character large object (CLOB)

The LOB datatype whose value is composed of character data corresponding to the database character set. A `CLOB` can be indexed and searched by the Oracle Text search engine.

CLOB

See character large object.

command line

The interface method in which the user enters in commands at the command interpreter prompt.

Common Gateway Interface (CGI)

The programming interfaces enabling Web servers to run other programs and pass their output to HTML pages, graphics, audio, and video sent to browsers.

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.

Content

The body of a resource is what you get when you treat the resource like a file and ask for its contents. Content is always an `XMLType`.

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 allowable 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.

empty element

An element without text content or child elements. It may only contain attributes and their values. Empty elements are of the form <name/> or <name></name> where there is no space between the tags.

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.

existsNode

The SQL operator that returns a `TRUE` or `FALSE` based upon the existence of an `XPath` within an `XMLType`.

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.

extract

The SQL operator that retrieves fragments of XML documents stored as `XMLType`.

Folder

A directory or node in the Oracle XML DB repository that contains or can contain a resource. A folder is also a resource.

Foldering

A feature in Oracle XML DB that allows content to be stored in a hierarchical structure of resources.

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.

function-based index

A database index that, when created, permits the results of known queries to be returned much more quickly.

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.

HTML

See Hypertext Markup Language.

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.

hypertext

The method of creating and publishing text documents in which users can navigate between other documents or graphics by selecting words or phrases designated as hyperlinks.

Hypertext Markup Language (HTML)

The markup language used to create the files sent to Web browsers and that serves as the basis of the World Wide Web. The next version of HTML will be called XHTML and will be an XML application.

Hypertext Transport Protocol (HTTP)

The application protocol used for transporting HTML files across the Internet between Web servers and browsers.

Hypertext Transport Protocol, Secure (HTTPS)

The use of Secure Sockets Layer (SSL) as a sub-layer under the regular HTTP application layer. Developed by Netscape.

iAS

See Internet Application Server.

IDE

See Integrated Development Environment.

iFS

See Internet File System.

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.

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.

interMedia

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 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 complied 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 XDK Java components, integrated for easy 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.

large object (LOB)

The class of SQL data type that is further divided into Internal LOBs and External LOBs. Internal LOBs include `BLOB`s, `CLOB`s, and `NCLOB`s while External LOBs include `BFILE`s. See also `BFILE`s, binary large object, character large object, national character large object.

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.

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.

namespace

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: `xmlns:xsl="http://www.w3.org/TR/WD-xsl"`.

national character large object (NCLOB)

The LOB datatype whose value is composed of character data corresponding to the database national character set.

NCLOB

See national character large object.

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.

An-tier

The designation for a computer communication network architecture that consists of one or more tiers made up of clients and servers. Typically two-tier systems are made up of one client level and one server level. A three-tier system utilizes two server tiers, typically a database server as one and a Web or application server along with a client tier.

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 (Oracle AS)

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 environmental variable that identifies the location of the Oracle database installation for use by applications.

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. Formerly known as Oracle Internet File System.

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.

Oracle9*i* JVM

The Java Virtual Machine that runs within the memory space of the Oracle database.

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.

parser

In XML, a software program that accepts as input an XML document and determines whether it is well-formed and, optionally, valid. The Oracle XML Parser supports both SAX and DOM interfaces.

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.

Pipeline Definition Language

W3C recommendation that enables you to describe the processing relations between XML resources.

PL/SQL

The Oracle procedural database language that extends SQL. It is used to create programs that can be run within the database.

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.

session

The active connection between two tiers.

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

See Simple Object Access Protocol.

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 XSL 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://otn.oracle.com:80/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.

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 referenced or included DTD.

W3C

See World Wide Web Consortium (W3C).

WebDAV

See World Wide Web distributed authoring and versioning.

Web Request Broker (WRB)

The cartridge within OAS 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)

An international industry consortium started in 1994 to develop standards for the World Wide Web. It is located at `http://www.w3c.org`.

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.

wrapper

The term describing a data structure or software that wraps around other data or software, typically to provide a generic or object interface.

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 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 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.

XML Gateway

A set of services that allows for easy integration with the Oracle E-Business Suite to create and consume XML messages triggered by business events.

XML Query

The on-going effort of the W3C to create a standard for the language and syntax to query XML documents.

XML Schema

W3C is creating a standard to enable 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. It can also extract the data from an XML document, then insert the data into a DB table, update a DB table, or delete corresponding data from a DB table.

XMLType

`XMLType` is an Oracle datatype that stores XML data using an underlying `CLOB` column or object-relational columns 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.

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

The designation used by the Oracle Servlet providing the ability to produce dynamic XML documents from one or more SQL queries and optionally transform the document in the server using an XSL stylesheet.

XSU

See XML SQL Utility.

Index