Contents

Preface ......................................................................................................................................................... xvii
  Audience ................................................................................................................................................ xvi
  Documentation Accessibility ................................................................................................................. xv
  Related Documents ............................................................................................................................ xvi
  Conventions ............................................................................................................................................ xv

What's New in Oracle C++ Call Interface? ...................................................................................... xix
  New Features for Oracle Database 11g Release 2 (11.2) ................................................................. xix
  New Features for Oracle Database 11g Release 1 (11.1) ................................................................. xix

1 Introduction to OCCI
  Overview of OCCI ................................................................................................................................. 1-1
    Benefits of OCCI ................................................................................................................................. 1-2
    Building an OCCI Application ........................................................................................................... 1-2
    Functionality of OCCI ......................................................................................................................... 1-3
    Procedural and Nonprocedural Elements .......................................................................................... 1-3
  Processing SQL Statements ................................................................................................................... 1-4
    Data Definition Language Statements ............................................................................................. 1-4
    Control Statements ............................................................................................................................ 1-5
    Data Manipulation Language Statements ......................................................................................... 1-5
    Queries .............................................................................................................................................. 1-5
  Overview of PL/SQL ............................................................................................................................... 1-5
  Special OCCI/SQL Terms ....................................................................................................................... 1-6
  Object Support .................................................................................................................................... 1-7
    Client-Side Object Cache ................................................................................................................... 1-8
    Run-time Environment for Objects .................................................................................................. 1-8
    Associative and Navigational Interfaces .......................................................................................... 1-8
    Interoperability with OCI ................................................................................................................... 1-9
    Metadata Class ................................................................................................................................. 1-9
    Object Type Translator Utility .......................................................................................................... 1-9
  Additional Support .............................................................................................................................. 1-10
    Building OCCI Demos ....................................................................................................................... 1-10
    OCCI on the Oracle Technology Network ......................................................................................... 1-10
2 Installation and Upgrading

Installing Oracle C++ Call Interface ................................................................................................................. 2-1
Upgrading Considerations ............................................................................................................................................. 2-1

Determining the Oracle Database Versions ............................................................................................................. 2-1
  Determining the Oracle Client Version During Compilation ................................................................................. 2-1
  Determining the Oracle Client and Server Versions at Run Time ........................................................................... 2-2

Instant Client ............................................................................................................................................................... 2-2
  Benefits of Instant Client ............................................................................................................................................ 2-2
  Installing Instant Client ............................................................................................................................................... 2-2
  Using Instant Client .................................................................................................................................................... 2-4
  Patching Instant Client Shared Libraries on UNIX ......................................................................................................... 2-4
  Regenerating the Data Shared Library and Zip Files ................................................................................................ 2-4
  Database Connection Names for Instant Client ........................................................................................................ 2-5
  Environment Variables for OCCI Instant Client ......................................................................................................... 2-5

Instant Client Light (English) ...................................................................................................................................... 2-6
  Globalization Settings for Instant Client Light (English) ............................................................................................. 2-6
  Using Instant Client Light (English) .......................................................................................................................... 2-7
  Installing Instant Client Light (English) ..................................................................................................................... 2-7

Using OCCI with Microsoft Visual C++ ...................................................................................................................... 2-8

3 Relational Programming

Connecting to a Database ............................................................................................................................................... 3-1
  Creating and Terminating an Environment .................................................................................................................... 3-1
  Opening and Closing a Connection ............................................................................................................................... 3-2

Pooling Connections ....................................................................................................................................................... 3-3
  Using Connection Pools ............................................................................................................................................... 3-3
  Stateless Connection Pooling ...................................................................................................................................... 3-5
  Database Resident Connection Pooling .......................................................................................................................... 3-9

Executing SQL DDL and DML Statements ................................................................................................................ 3-12
  Creating a Statement Object ....................................................................................................................................... 3-12
  Creating a Statement Object that Executes SQL Commands .................................................................................... 3-12
  Reusing the Statement Object ................................................................................................................................... 3-13
  Terminating a Statement Object .................................................................................................................................. 3-13

Types of SQL Statements in the OCCI Environment ................................................................................................. 3-13
  Standard Statements .................................................................................................................................................... 3-13
  Parameterized Statements ......................................................................................................................................... 3-13
  Callable Statements .................................................................................................................................................... 3-14
  Streamed Reads and Writes ....................................................................................................................................... 3-15
  Modifying Rows Iteratively ....................................................................................................................................... 3-18

Executing SQL Queries .................................................................................................................................................. 3-19
  Using the Result Set ..................................................................................................................................................... 3-19
  Specifying the Query .................................................................................................................................................... 3-20
  Optimizing Performance by Setting Prefetch Count .................................................................................................. 3-20

Executing Statements Dynamically ............................................................................................................................. 3-21
  Statement Status Definitions ....................................................................................................................................... 3-21

Committing a Transaction ............................................................................................................................................. 3-23

Caching Statements ....................................................................................................................................................... 3-24
4 Object Programming

Overview of Object Programming ......................................................................................... 4-1
Working with Objects in OCCI ............................................................................................... 4-2
   Persistent Objects ................................................................................................................. 4-2
   Transient Objects ................................................................................................................. 4-3
   Values ................................................................................................................................. 4-4

Representing Objects in C++ Applications ........................................................................ 4-4
   Creating Persistent and Transient Objects ......................................................................... 4-4
   Creating Object Representations using the OTT Utility ..................................................... 4-5

Developing an OCCI Object Application ........................................................................... 4-6
   Basic Object Program Structure ....................................................................................... 4-6
   Basic Object Operational Flow ......................................................................................... 4-7

Migrating C++ Applications Using OCCI ........................................................................... 4-9

Overview of Associative Access ............................................................................................ 4-10
   Using SQL to Access Objects ............................................................................................. 4-10
   Inserting and Modifying Values ......................................................................................... 4-10

Overview of Navigational Access ....................................................................................... 4-10
   Retrieving an Object Reference (REF) from the Database Server .................................. 4-11
   Pinning an Object .............................................................................................................. 4-11
   Manipulating Object Attributes ....................................................................................... 4-12
   Marking Objects and Flushing Changes ......................................................................... 4-12
   Marking an Object as Modified (Dirty) ........................................................................... 4-12
   Recording Changes in the Database .................................................................................. 4-12
   Collecting Garbage in the Object Cache ....................................................................... 4-13
   Ensuring Transactional Consistency of References ......................................................... 4-14

Overview of Complex Object Retrieval ............................................................................ 4-14
   Retrieving Complex Objects ............................................................................................. 4-14
   Prefetching Complex Objects ......................................................................................... 4-16

Working with Collections ..................................................................................................... 4-16
   Fetching Embedded Objects .............................................................................................. 4-17
   Nullness ............................................................................................................................ 4-17

Using Object References ....................................................................................................... 4-18

Deleting Objects from the Database ..................................................................................... 4-18

Type Inheritance ................................................................................................................ 4-18
   Substitutability ................................................................................................................ 4-19
   NOT INSTANTIABLE Types and Methods ..................................................................... 4-20
   OCCI Support for Type Inheritance ................................................................................ 4-20
   OTT Support for Type Inheritance .................................................................................. 4-20

A Sample OCCI Application ............................................................................................... 4-21

5 Data Types

Overview of Oracle Data Types .......................................................................................... 5-1
   OCCI Type and Data Conversion ..................................................................................... 5-1
# External Data Types

## Data Conversions

- Data Conversions for LOB Data Types
- Data Conversions for Date, Timestamp, and Interval Data Types

# Metadata

## Overview of Metadata

## Describing Database Metadata

- Metadata Code Examples

## Attribute Reference

- Parameter Attributes
- Table and View Attributes
- Procedure, Function, and Subprogram Attributes
- Package Attributes
- Type Attributes
- Type Attribute Attributes
- Type Method Attributes
- Collection Attributes
- Synonym Attributes
- Sequence Attributes
- Column Attributes
- Argument and Result Attributes
- List Attributes
- Schema Attributes
- Database Attributes

# Programming with LOBs

## Overview of LOBs

- Introducing Internal LOBs
- Introducing External LOBs
- Storing LOBs

## Creating LOBs in OCCI Applications

## Restricting the Opening and Closing of LOBs

## Reading and Writing LOBs

- Reading LOBs
- Writing LOBs
- Enhancing the Performance of LOB Reads and Writes
- Updating LOBs
- Reading and Writing Multiple LOBs

## Using Objects with LOB Attributes

## Using SecureFiles

- Using SecureFile Compression
- Using SecureFiles Encryption
11 Oracle XA Library

Application Development with XA and OCCI ............................................................................. 11-1
APIs for XA Support .................................................................................................................. 11-2

12 Optimizing Performance of OCCI Applications

Transparent Application Failover ................................................................................................. 12-1
Using Transparent Application Failover ....................................................................................... 12-2
Objects and Transparent Application Failover ............................................................................. 12-3
Connection Pooling and Transparent Application Failover ......................................................... 12-3
Connection Sharing ...................................................................................................................... 12-5
Introduction to Thread Safety ....................................................................................................... 12-6
Implementing Thread Safety ......................................................................................................... 12-6
Serialization .................................................................................................................................. 12-7
Operating System Considerations ................................................................................................. 12-8
Application-Managed Data Buffering ............................................................................................ 12-8
setDataBuffer() Method ............................................................................................................. 12-9
executeArrayUpdate() Method .................................................................................................. 12-10
Array Fetch Using next() Method ............................................................................................... 12-11
Modifying Rows Iteratively .......................................................................................................... 12-11
Run-time Load Balancing of the Stateless Connection Pool ......................................................... 12-12
API Support .................................................................................................................................. 12-12
Fault Diagnosability ....................................................................................................................... 12-12
ADR Base Location ....................................................................................................................... 12-13
Using ADRCI ................................................................................................................................ 12-14
Controlling ADR Creation and Disabling Fault Diagnosability ................................................... 12-16
Client Result Cache ....................................................................................................................... 12-16

13 OCCI Application Programming Interface

OCCI Classes and Methods ........................................................................................................... 13-2
Common OCCI Constants ............................................................................................................. 13-6
Agent Class ................................................................................................................................... 13-7
AnyData Class ............................................................................................................................... 13-10
BatchSQLException Class ............................................................................................................. 13-19
<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bfile Class</td>
<td>13-20</td>
</tr>
<tr>
<td>Blob Class</td>
<td>13-26</td>
</tr>
<tr>
<td>Bytes Class</td>
<td>13-35</td>
</tr>
<tr>
<td>Clob Class</td>
<td>13-38</td>
</tr>
<tr>
<td>Connection Class</td>
<td>13-49</td>
</tr>
<tr>
<td>ConnectionPool Class</td>
<td>13-62</td>
</tr>
<tr>
<td>Consumer Class</td>
<td>13-67</td>
</tr>
<tr>
<td>Date Class</td>
<td>13-75</td>
</tr>
<tr>
<td>Environment Class</td>
<td>13-85</td>
</tr>
<tr>
<td>IntervalDS Class</td>
<td>13-97</td>
</tr>
<tr>
<td>IntervalYM Class</td>
<td>13-107</td>
</tr>
<tr>
<td>Listener Class</td>
<td>13-116</td>
</tr>
<tr>
<td>Map Class</td>
<td>13-118</td>
</tr>
<tr>
<td>Message Class</td>
<td>13-120</td>
</tr>
<tr>
<td>MetaData Class</td>
<td>13-128</td>
</tr>
<tr>
<td>NotifyResult Class</td>
<td>13-142</td>
</tr>
<tr>
<td>Number Class</td>
<td>13-143</td>
</tr>
<tr>
<td>PObject Class</td>
<td>13-162</td>
</tr>
<tr>
<td>Producer Class</td>
<td>13-168</td>
</tr>
<tr>
<td>Ref Class</td>
<td>13-173</td>
</tr>
<tr>
<td>RefAny Class</td>
<td>13-178</td>
</tr>
<tr>
<td>ResultSet Class</td>
<td>13-181</td>
</tr>
<tr>
<td>SQLException Class</td>
<td>13-200</td>
</tr>
<tr>
<td>StatelessConnectionPool Class</td>
<td>13-203</td>
</tr>
<tr>
<td>Statement Class</td>
<td>13-213</td>
</tr>
<tr>
<td>Stream Class</td>
<td>13-254</td>
</tr>
<tr>
<td>Subscription Class</td>
<td>13-257</td>
</tr>
<tr>
<td>Timestamp Class</td>
<td>13-264</td>
</tr>
</tbody>
</table>

**Index**
List of Examples

1-1  Creating a Table ................................................................. 1-4
1-2  Specifying Access to a Table.................................................. 1-4
1-3  Creating an Object Table....................................................... 1-4
1-4  Inserting Data Through Input Bind Variables.......................... 1-5
1-5  Inserting Objects into the Oracle Database.............................. 1-5
1-6  Using the Simple SELECT Statement...................................... 1-5
1-7  Using the SELECT Statement with Input Variables .................. 1-5
1-8  Using PL/SQL to Obtain an Output Variable........................... 1-6
1-9  Using PL/SQL to Insert Partial Records into Placeholders......... 1-6
1-10 Using SQL to Extract Partial Records................................... 1-6
2-1  How to Determine the Major Client Version and Set Performance Features ........................................ 2-1
2-2  How to Regenerate the Data Shared Library Files.................... 2-5
2-3  How to set the ORA_TZFILE Environment Variable.................. 2-6
2-4  Installing Instant Client Light (English) through Oracle Universal Installer ........................................... 2-8
3-1  How to Create an OCCI Environment....................................... 3-1
3-2  How to Terminate an OCCI Environment................................. 3-1
3-3  How to Use Environment Scope with Blob Objects.................... 3-2
3-4  How to Create an Environment and then a Connection to the Database .................................................. 3-2
3-5  How to Terminate a Connection to the Database and the Environment .................................................. 3-3
3-6  The createConnectionPool() Method ...................................... 3-3
3-7  How to Create a Connection Pool .......................................... 3-4
3-8  The createProxyConnection() Method ...................................... 3-4
3-9  How to Use a StatelessConnectionPool ................................... 3-6
3-10 How to Create and Use a Homogeneous Stateless Connection Pool.................................................. 3-6
3-11 How to Create and Use a Heterogeneous Stateless Connection Pool.................................................. 3-7
3-12 How to Administer the Database Resident Connection Pools ......... 3-10
3-13 How to Get a Connection from a Database Resident Connection Pool .................................................. 3-11
3-14 Using Client-Side Pool and Server-Side Pool........................... 3-11
3-15 How to Create a Statement.................................................... 3-12
3-16 How to Create a Database Table Using the executeUpdate() Method .................................................. 3-12
3-17 How to Add Records Using the executeUpdate() Method ............ 3-12
3-18 How to Specify a SQL Statement Using the setSQL() Method ......... 3-13
3-19 How to Reset a SQL Statement Using the setSQL() Method .......... 3-13
3-20 How to Terminate a Statement Using the terminateStatement() Method .................................................. 3-13
3-21 How to Use setxxx() Methods to Set Individual Column Values .................................................. 3-14
3-22 How to Specify the IN Parameters of a PL/SQL Stored Procedure .................................................. 3-14
3-23 How to Specify OUT Parameters of a PL/SQL Stored Procedure .................................................. 3-14
3-24 How to Bind Data in a Streaming Mode ................................... 3-16
3-25 How to Fetch Data in a Streaming Mode Using PL/SQL ................. 3-17
3-26 How to Read and Write with Multiple Streams......................... 3-17
3-27 How to Execute an Iterative Operation ................................... 3-19
3-28 How to Fetch Data in Streaming Mode Using ResultSet .............. 3-20
3-29 Statement Caching without Connection Pooling ....................... 3-24
3-30 Statement Caching with Connection Pooling ........................... 3-25
4-1  Creating Standalone Objects ................................................. 4-3
4-2  Creating Embedded Objects.................................................. 4-3
4-3  Two Methods for Operator new() in the Object Type Translator Utility .................................................. 4-3
4-4  How to Dynamically Create a Transient Object ......................... 4-4
4-5  How to Create a Transient Object as a Local Variable ................. 4-4
4-6  How to Create a Persistent Object........................................... 4-4
4-7  How to Create a Transient Object ........................................... 4-5
4-8  How to Declare a Custom Type in the Database ........................ 4-5
4-9  How to Create a VARRAY Collection ........................................ 4-16
4-10 OTT Support Inheritance ....................................................... 4-20
4–11  Listing of demo2.sql for a Sample OCCI Application ................................................. 4-21
4–12  Listing of demo2.typ for a Sample OCCI Application .................................................. 4-22
4–13  Listing of OTT Command that Generates Files for a Sample OCCI Application .... 4-22
4–14  Listing of mappings.h for a Sample OCCI Application ............................................ 4-22
4–15  Listing of mappings.cpp for a Sample OCCI Application ........................................ 4-22
4–16  Listing of demo2.h for a Sample OCCI Application ................................................. 4-22
4–17  Listing of demo2.cpp for a Sample OCCI Application .............................................. 4-26
4–18  Listing of myDemo.h for a Sample OCCI Application ............................................... 4-34
4–19  Listing for myDemo.cpp for a Sample OCCI Application ........................................... 4-35
4–20  Listing of main.cpp for a Sample OCCI Application ................................................ 4-37
5–1   Definition of the BDOUBLE Data Type ................................................................. 5-7
5–2   Definition of the BFLOAT Data Type ..................................................................... 5-7
6–1   How to Obtain Metadata About Attributes of a Simple Database Table .............. 6-3
6–2   How to Obtain Metadata from a Column Containing User-Defined Types ........... 6-4
6–3   How to Obtain Object Metadata from a Reference ................................................ 6-5
6–4   How to Obtain Metadata About a Select List from a ResultSet Object ................. 6-5
7–1   How to Read Non-Streamed BLOBs .......................................................... 7-4
7–2   How to Read Non-Streamed BFILES ....................................................... 7-4
7–3   How to Read Streamed BLOBs ............................................................................. 7-5
7–4   How to Write Non-Streamed BLOBs ................................................................. 7-5
7–5   How to Write Streamed BLOBs ............................................................................. 7-6
7–6   Updating a CLOB Value ....................................................................................... 7-7
7–7   Updating a BFILE Value ....................................................................................... 7-7
7–8   How to Use a Persistent Object with a BLOB Attribute ......................................... 7-8
7–9   How to Use a Persistent Object with a BFILE Attribute .......................................... 7-9
8–1   How to Use the OTT Utility .................................................................................... 8-2
8–2   Object Creation Statements of the OTT Utility ..................................................... 8-2
8–3   How to Invoke the OTT Utility to Generate C++ Classes ..................................... 8-4
8–4   How to use the SCHEMA_NAMES Parameter in OTT Utility .................................. 8-8
8–5   How to Define a Schema for Unicode Support in OTT ........................................... 8-10
8–6   How to Use UNICODE=ALL Parameter in OTT .................................................... 8-10
8–7   How to Use UNICODE=ONLYCHAR Parameter in OTT ....................................... 8-10
8–8   How to Create a User Defined INTYPE File Using the OTT Utility ..................... 8-12
8–9   Listing of ott95a.h ................................................................................................. 8-16
8–10  Listing of ott95b.h ................................................................................................. 8-16
8–11  How to Represent Object Attributes Using the OTT Utility ................................ 8-18
8–12  How to Map Object Data Types Using the OTT Utility ......................................... 8-19
8–13  OUTTYPE File Generated by the OTT Utility ....................................................... 8-23
8–14  How to Generate C++ Classes Using the OTT Utility ........................................... 8-25
8–15  How to Extend C++ Classes Using the OTT Utility ................................................ 8-27
8–16  How to Add User Code to a Header File Using OTT Utility ..................................... 8-29
8–17  How to Add User Code to the Source File Using the OTT Utility ....................... 8-30
9–1   How to Use Globalization and Unicode Support .................................................... 9-2
9–2   Using wstring Data Type ....................................................................................... 9-2
9–3   Binding UTF8 Data Using the string Data Type .................................................... 9-3
9–4   Binding UTF16 Data Using the UString Data Type ............................................... 9-3
9–5   Using CLOB and NCLOB Data Types ...................................................................... 9-3
10–1  Creating an Agent ................................................................................................. 10-3
10–2  Setting the Agent on the Consumer ..................................................................... 10-3
10–3  Creating an AnyData Message with a String Payload ........................................ 10-4
10–4  Determining the Type of the Payload in an AnyData Message ............................ 10-4
10–5  Creating an User-defined Payload ......................................................................... 10-4
10–6  Specifying the Correlation identifier ..................................................................... 10-5
10–7  Specifying the Sender identifier ............................................................................ 10-5
10–8  Specifying the Delay and Expiration times of the message ................................. 10-5
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>The OCCI Development Process</td>
<td>1-2</td>
</tr>
<tr>
<td>4-1</td>
<td>Basic Object Operational Flow</td>
<td>4-7</td>
</tr>
<tr>
<td>8-1</td>
<td>The OTT Utility with OCCI</td>
<td>8-24</td>
</tr>
</tbody>
</table>
13–21 Fields of IntervalYM Class................................................................. 13-107
13–22 Summary of IntervalYM Methods...................................................... 13-108
13–23 Summary of Listener Methods............................................................ 13-116
13–24 Summary of Map Methods............................................................... 13-118
13–25 Enumerated Values Used by Message Class....................................... 13-120
13–26 Summary of Message Methods......................................................... 13-120
13–27 Enumerated Values Used by MetaData Class....................................... 13-128
13–28 Summary of MetaData Methods....................................................... 13-136
13–29 Summary of NotifyResult Methods.................................................... 13-142
13–30 Summary of Number Methods......................................................... 13-144
13–31 Enumerated Values Used by PObject Class........................................ 13-162
13–32 Summary of PObject Methods......................................................... 13-162
13–33 Enumerated Values Used by Producer Class....................................... 13-168
13–34 Summary of Producer Methods....................................................... 13-168
13–35 Enumerated Values Used by Ref Class................................................ 13-173
13–36 Summary of Ref Methods............................................................... 13-173
13–37 Summary of RefAny Methods........................................................... 13-178
13–38 Enumerated Values Used by ResultSet Class...................................... 13-181
13–39 Summary of ResultSet Methods...................................................... 13-181
13–40 Summary of SQLException............................................................... 13-200
13–41 Enumerated Values Used by StatelessConnectionPool Class............... 13-203
13–42 Summary of StatelessConnectionPool Methods.................................. 13-203
13–43 Enumerated Values used by the Statement Class............................... 13-213
13–44 Statement Methods........................................................................... 13-213
13–45 Enumerated Values Used by Stream Class......................................... 13-254
13–46 Summary of Stream Methods........................................................... 13-254
13–47 Enumerated Values Used by Subscription Class.................................. 13-257
13–48 Summary of Subscription Methods................................................... 13-257
13–49 Fields of Timestamp and Their Legal Ranges.................................... 13-264
13–50 Summary of Timestamp Methods.................................................... 13-264
Preface

The Oracle C++ Call Interface (OCCI) is an application programming interface (API) that allows applications written in C++ to interact with one or more Oracle database servers. OCCI gives your programs the ability to perform the full range of database operations that are possible with an Oracle database server, including SQL statement processing and object manipulation.

Audience

The Oracle C++ Call Interface Programmer’s Guide is intended for programmers, system analysts, project managers, and other Oracle users who perform, or are interested in learning about, the following tasks:

- Design and develop database applications in the Oracle environment.
- Convert existing database applications to run in the Oracle environment.
- Manage the development of database applications.

To use this document, you need a basic understanding of object-oriented programming concepts, familiarity with the use of Structured Query Language (SQL), and a working knowledge of application development using C++.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. 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.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Related Documents

For more information, see these Oracle resources:

- Oracle C++ Call Interface product information page for white papers, additional examples, and so on, at http://www.oracle.com/technology/tech/oci/occi/
- Discussion forum for all Oracle C++ Call Interface related information is at http://forums.oracle.com/forums/forum.jspa?forumID=168
- Demos at $ORACLE_HOME/rdbms/demo
- Oracle Database Concepts
- Oracle Database SQL Language Reference
- Oracle Database Object-Relational Developer’s Guide
- Oracle Database SecureFiles and Large Objects Developer’s Guide
- Oracle Database New Features Guide
- Oracle Call Interface Programmer’s Guide
- Oracle Database Administrator’s Guide
- Oracle Streams Advanced Queuing User’s Guide
- Oracle Database Globalization Support Guide
- Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to Oracle Database Sample Schemas for information on how these schemas were created and how you can use them yourself.

Conventions

The following text conventions are used in this document:

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

This section describes new features in Oracle C++ Call Interface Programmer’s Guide and supplies pointers to additional information.

New Features for Oracle Database 11g Release 2 (11.2)

The following OCCI feature is new to the Oracle Database 11g Release 2:

**SecureFiles Content Types**

SecureFiles can now be associated with a content type. Content types can be set using `setContentType()` and retrieved with `getContentType()` methods. Note that content types are limited to 128 bytes. Primary documentation for this feature is in the Oracle Database SecureFiles and Large Objects Developer’s Guide.

This feature adds new methods to the Blob Class:
- `getContentType()` on page 13-29
- `setContentType()` on page 13-32

This feature adds new methods to the Clob Class:
- `getContentType()` on page 13-41
- `setContentType()` on page 13-46

New Features for Oracle Database 11g Release 1 (11.1)

The following OCCI features are new to the Oracle Database 11g Release 1:

**Database Resident Connection Pooling**

Documentation for this feature is in section "Database Resident Connection Pooling" on page 3-9.

"Administrating Database Resident Connection Pools" on page 3-9 describes the responsibilities of a SYSDBA user in activating and maintaining this feature, while "Using Database Resident Connection Pools" on page 3-10 discusses the programmatic steps.

This feature adds new versions of this method to Environment Class:
- `createConnection()` on page 13-86, two new versions

This feature adds new versions of these methods to StatelessConnectionPool Class:
- `getConnection()` on page 13-206, six new versions
- `getProxyConnection()` on page 13-208, four new versions

**SecureFile Compression, Encryption, and Deduplication**

This feature is thoroughly described in the *Oracle Database SecureFiles and Large Objects Developer’s Guide*. Information specific to OCCI is in section "Using SecureFiles" on page 7-9 of Chapter 7, “Programming with LOBs”. Section "SecureFiles LOB Types and Constants" on page 7-10 describes the new LOB attributes, while Chapter 13, "OCCI Application Programming Interface" documents the new and changed APIs for the OCCI interface.

Section "SecureFiles LOB Types and Constants" lists two new attribute types for LOBs, and adds more options to an existing type. These are all listed in the following tables:

- Table 7–1, "Values of Type LobOptionType" on page 7-10
- Table 7–2, "Values of Type LobOptionValue" on page 7-10

The following new methods are added to the *Blob Class*:

- `getOptions()` on page 13-29
- `setOptions()` on page 13-32

The following new methods are added to the *Clob Class*:

- `getOptions()` on page 13-42
- `setOptions()` on page 13-47

In addition, the behavior of the following methods changed for both *Blob Class* and *Clob Class*: `getChunkSize()` and `copy()`.

**Run-time Load Balancing of the Stateless Connection Pool**

Information about this feature is in section "Run-time Load Balancing of the Stateless Connection Pool" on page 12-12.

New APIs for this feature are described in Chapter 13, "OCCI Application Programming Interface":

- A *StatelessConnectionPool Class* attribute `PoolType`, defined on page 13-203, has a new option, `NO_RLB`, used to turn off this feature.

**Fault Diagnosability**

Information about this feature is in section "Fault Diagnosability" on page 12-12.

**Client Result Cache**

Documentation for this feature is in section "Client Result Cache" on page 12-16.
This chapter provides an overview of Oracle C++ Call Interface (OCCI) and introduces terminology used in discussing OCCI. You are provided with the background information needed to develop C++ applications that run in an Oracle environment.

This chapter contains these topics:

- Overview of OCCI
- Processing SQL Statements
- Overview of PL/SQL
- Special OCCI/SQL Terms
- Object Support
- Additional Support

Overview of OCCI

Oracle C++ Call Interface (OCCI) is an Application Programming Interface (API) that provides C++ applications access to data in an Oracle database. OCCI enables C++ programmers to use the full range of Oracle database operations, including SQL statement processing and object manipulation.

OCCI provides for:

- High performance applications through the efficient use of system memory and network connectivity
- Scalable applications that can service an increasing number of users and requests
- Comprehensive support for application development by using Oracle database objects, including client-side access to Oracle database objects
- Simplified user authentication and password management
- n-tiered authentication
- Consistent interfaces for dynamic connection management and transaction management in two-tier client/server environments or multitiered environments
- Encapsulated and opaque interfaces

OCCI provides a library of standard database access and retrieval functions in the form of a dynamic run-time library (OCCI classes) that can be linked in a C++ application at run time. This eliminates the requirement to embed SQL or PL/SQL within third-generation language (3GL) programs.

This section discusses the following topics:
Benefits of OCCI

OCCI provides these significant advantages over other methods of accessing an Oracle database:

- Leverages C++ and the Object Oriented Programming paradigm
- Is easy to use
- Is easy to learn for those familiar with JDBC
- Has a navigational interface to manipulate database objects of user-defined types as C++ class instances

Note that OCCI does not support nonblocking mode.

Building an OCCI Application

As Figure 1-1 shows, you compile and link an OCCI program in the same way that you compile and link an application that does not use the database.
Oracle supports most popular third-party compilers. The details of linking an OCCI program vary from system to system. On some platforms, it may be necessary to include other libraries, in addition to the OCCI library, to properly link your OCCI programs.

See Also: Your operating system-specific Oracle documentation and the Oracle Database Installation Guide for more information about compiling and linking an OCCI application for your specific platform

Functionality of OCCI

OCCI provides the following functionality:

- APIs to design scalable, multithreaded applications that can support large numbers of users securely
- SQL access functions, for managing database access, processing SQL statements, and manipulating objects retrieved from an Oracle database server
- Data type mapping and manipulation functions, for manipulating data attributes of Oracle types
- Advanced Queuing for message management
- XA compliance for distributed transaction support
- Statement caching of SQL and PL/SQL queries
- Connection pooling for managing multiple connections
- Globalization and Unicode support to customize applications for international and regional language requirement
- Object Type Translator Utility
- Transparent Application Failover support

Procedural and Nonprocedural Elements

Oracle C++ Call Interface (OCCI) enables you to develop scalable, multithreaded applications on multilayered architectures that combine nonprocedural data access power of structured query language (SQL) with procedural capabilities of C++.

In a nonprocedural language program, the set of data to be operated on is specified, but what operations may performed, or how the operations can be carried out, is not specified. The nonprocedural nature of SQL makes it an easy language to learn and use to perform database transactions. It is also the standard language used to access and manipulate data in modern relational and object-relational database systems.

In a procedural language program, the execution of most statements depends on previous or subsequent statements and on control structures, such as loops or conditional branches, which are not available in SQL. The procedural nature of these languages makes them more complex than SQL, but it also makes them very flexible and powerful.

The combination of both nonprocedural and procedural language elements in an OCCI program provides easy access to an Oracle database in a structured programming environment.

OCCI supports all SQL data definition, data manipulation, query, and transaction control facilities that are available through an Oracle database server. For example, an OCCI program can run a query against an Oracle database. The queries can require the program to supply data to the database by using input (bind) variables, as follows:
SELECT name FROM employees WHERE employee_id = :empnum

In this SQL statement, empnum is a placeholder for a value that is supplied by the application.

In an OCCI application, you can also take advantage of PL/SQL, Oracle's procedural extension to SQL. The applications you develop can be more powerful and flexible than applications written in SQL alone. OCCI also provides facilities for accessing and manipulating objects in an Oracle database server.

Processing SQL Statements

One of the main tasks of an OCCI application is to process SQL statements. Different types of SQL statements require different processing steps in your program. It is important to take this into account when coding your OCCI application. Oracle recognizes several types of SQL statements:

- Data Definition Language Statements (DDL)
- Control Statements
- Data Manipulation Language Statements (DML)
- Queries

Data Definition Language Statements

Data Definition Language (DDL) statements manage schema objects in the database. These statements create new tables, drop old tables, and establish other schema objects. They also control access to schema objects. Example 1–1 illustrates how to create a table, and Example 1–2 shows how to grant and revoke privileges on this table.

**Example 1–1 Creating a Table**

```sql
CREATE TABLE employee_information (
    employee_id NUMBER(6),
    ssn NUMBER(9),
    dependents NUMBER(1),
    mail_address VARCHAR(60))
```

**Example 1–2 Specifying Access to a Table**

```sql
GRANT UPDATE, INSERT, DELETE ON employee_information TO donna
REVOKE UPDATE ON employee_information FROM jamie
```

DDL statements also allow you to work with objects in the Oracle database, as in Example 1–3, which illustrates how to create an object table.

**Example 1–3 Creating an Object Table**

```sql
CREATE TYPE person_info_type AS OBJECT {
    employee_id NUMBER(6),
    ssn NUMBER(9),
    dependents NUMBER(1),
    mail_address VARCHAR(60)}

CREATE TABLE person_info_table OF person_info_type
```
Control Statements

OCCI applications treat transaction control, connection control, and system control statements (for example, DML statements).

See Also: Oracle Database SQL Language Reference for information about control statements.

Data Manipulation Language Statements

Data Manipulation Language (DML) statements can change data in database tables. For example, DML statements insert new rows into a table, update column values in existing rows, delete rows from a table, lock a table in the database, and explain the execution plan for a SQL statement.

DML statements may require an application to supply data to the database by using input bind variables, as in Example 1–4. This statement can be executed several times with different bind values, or several rows can be added through array insert in a single round-trip to the server.

Example 1–4 Inserting Data Through Input Bind Variables

\[
\text{INSERT INTO departments VALUES}(:1,:2,:3)
\]

DML statements also enable you to work with objects in the Oracle Database, as in Example 1–5, which shows the insertion of an instance of a type into an object table.

Example 1–5 Inserting Objects into the Oracle Database

\[
\text{INSERT INTO person_info_table}
\text{VALUES (person_info_type('450987','123456789','3','146 Winfield Street'))}
\]

Queries

Queries are statements that retrieve data from tables in a database. A query can return zero, one, or many rows of data. All queries begin with the SQL keyword SELECT, as in Example 1–6:

Example 1–6 Using the Simple SELECT Statement

\[
\text{SELECT department_name FROM departments}
\text{WHERE department_id = 30}
\]

Queries can require the program to supply data to the database server by using input bind variables, as in Example 1–7:

Example 1–7 Using the SELECT Statement with Input Variables

\[
\text{SELECT first_name, last_name}
\text{FROM employees}
\text{WHERE employee_id = :emp_id}
\]

In this SQL statement, \text{emp_id} is a placeholder for a value that is supplied by the application.

Overview of PL/SQL

PL/SQL is Oracle’s procedural extension to the SQL language. PL/SQL processes tasks that are more complicated than simple queries and SQL data manipulation language
statements. PL/SQL allows several constructs to be grouped into a single block and executed as a unit. Among these are the following constructs:

- One or more SQL statements
- Variable declarations
- Assignment statements
- Procedural control statements (IF...THEN...ELSE statements and loops)
- Exception handling

In addition to calling PL/SQL stored procedures from an OCCI program, you can use PL/SQL blocks in your OCCI program to perform the following tasks:

- Call other PL/SQL stored procedures and stored functions.
- Combine procedural control statements with several SQL statements, to be executed as a unit.
- Access special PL/SQL features such as records, tables, cursor FOR loops, and exception handling.
- Use cursor variables.
- Access and manipulate objects in an Oracle database.

A PL/SQL procedure or function can also return an output variable. This is called an out bind variable, as in Example 1–8:

**Example 1–8 Using PL/SQL to Obtain an Output Variable**

```sql
GET_EMPLOYEE_NAME(:1, :2);
```

Here, the first parameter is an input variable that provides the ID number of an employee. The second parameter, or the out bind variable, contains the return value of employee name.

PL/SQL can also be used to issue a SQL statement to retrieve values from a table of employees, given a particular employee number. Example 1–9 demonstrates the use of placeholders in PL/SQL statements.

**Example 1–9 Using PL/SQL to Insert Partial Records into Placeholders**

```sql
SELECT last_name, first_name, salary, commission_pct INTO :emp_last, :emp_first, :sal, :comm FROM employees WHERE employee_id = :emp_id;
```

Note that the placeholders in this statement are not PL/SQL variables. They represent input and output parameters passed to and from the database server when the statement is processed. These placeholders must be specified in your program.

**Special OCCI/SQL Terms**

This guide uses special terms to refer to the different parts of a SQL statement. Consider Example 1–10:

**Example 1–10 Using SQL to Extract Partial Records**

```sql
SELECT first_name, last_name, email FROM employees
```
WHERE department_id = 80
AND commission_pct > :base;

This example contains these parts:

- A SQL command: SELECT
- Three select-list items: first_name, last_name, and email
- A table name in the FROM clause: employees
- Two column names in the WHERE clause: department_id and commission_pct
- A numeric input value in the WHERE clause: 40
- A placeholder for an input bind variable in the WHERE clause: :base

When you develop your OCCI application, you call routines that specify to the database server the value of, or reference to, input and output variables in your program. In this guide, specifying the placeholder variable for data is called a bind operation. For input variables, this is called an in bind operation. For output variables, this is called an out bind operation.

Object Support

OCCI has facilities for working with object types and objects. An object type is a user-defined data structure representing an abstraction of a real-world entity. For example, the database might contain a definition of a person object. That object type might have attributes, such as first_name, last_name, and age, which represent a person's identifying characteristics.

The object type definition serves as the basis for creating objects, which represent instances of the object type. By using the object type as a structural definition, a person object could be created with the attributes John, Bonivento, and 30. Object types may also contain methods, or programmatic functions that represent the behavior of that object type.

OCCI provides a comprehensive API for programmers seeking to use the Oracle database server’s object capabilities. These features can be divided into several major categories, which are discussed in subsequent topics:

- Client-Side Object Cache
- Run-time Environment for Objects
- Associative and Navigational Interfaces
- Interoperability with OCI
- Metadata Class
- Object Type Translator Utility (OTT) utility, which maps internal Oracle schema information to client-side language bind variables

See Also:
- Oracle Database Concepts
- Oracle Database Object-Relational Developer’s Guide for a more detailed explanation of object types and objects
Client-Side Object Cache

The object cache is a client-side memory buffer that provides lookup and memory management support for objects. It stores and tracks objects which have been fetched by an OCCI application from the server to the client side. The client-side object cache is created when the OCCI environment is initialized in object mode. Multiple applications running against the same server have their own object cache. The client-side object cache tracks the objects that are currently in memory, maintains references to objects, manages automatic object swapping and tracks the meta-attributes or type information about objects. The client-side object cache provides the following benefits:

- Improved application performance by reducing the number of client/server round-trips required to fetch and operate on objects
- Enhanced scalability by supporting object swapping from the client-side cache
- Improved concurrency by supporting object-level locking
- Automatic garbage collection when cache thresholds are exceeded

Run-time Environment for Objects

OCCI provides a run-time environment for objects that offers a set of methods for managing how Oracle objects are used on the client side. These methods provide the necessary functionality for performing these tasks:

- Connecting to an Oracle database server to access its object functionality
- Allocating the client-side object cache and tuning its parameters
- Retrieving error and warning messages
- Controlling transactions that access objects in the database
- Associatively accessing objects through SQL
- Describing a PL/SQL procedure or function whose parameters or result are of Oracle object type

Associative and Navigational Interfaces

Applications that use OCCI can access objects in the database through several types of interfaces:

- SQL SELECT, INSERT, and UPDATE statements
- C++ pointers and references to access objects in the client-side object cache by traversing the corresponding references

OCCI provides a set of methods to support object manipulation by using SQL SELECT, INSERT, and UPDATE statements. To access Oracle objects, these SQL statements use a consistent set of steps as if they were accessing relational tables. OCCI provides methods to access objects by using SQL statements for:

- Binding object type instances and references as input and output variables of SQL statements and PL/SQL stored procedures
- Executing SQL statements that contain object type instances and references
- Fetching object type instances and references
- Retrieving column values from a result set as objects
- Describing a select-list item of an Oracle object type
OCCI provides a seamless interface for navigating objects, enabling you to manipulate database objects in the same way that you would operate on transient C++ objects. You can dereference the overloaded arrow (\rightarrow) operator on an object reference to transparently materialize the object from the database into the application space.

**Interoperability with OCI**

The OCCI application can retrieve the underlying OCI handles and descriptors by calling `getOCIxxx()` methods on the OCCI class instances. These handles can be used to call OCI functions.

Note that the application must be aware that if any properties are changed on the OCI handles, the corresponding OCCI instances may or may not reflect this.

This interoperability between OCCI and OCI is not supported if the application uses OCI functions for any object-related functionality.

**Metadata Class**

Each Oracle data type is represented in OCCI by a C++ class. The class exposes the behavior and characteristics of the data type by overloaded operators and methods. For example, the Oracle data type `NUMBER` is represented by the `Number` class.

OCCI provides a metadata class that enables you to retrieve metadata describing database objects, including object types.

**Object Type Translator Utility**

The Object Type Translator (OTT) utility translates schema information about Oracle object types into client-side language bindings. That is, OTT translates object type information into declarations of host language variables, such as structures and classes. OTT takes an `intype` file that contains information about Oracle database schema objects as input. OTT generates an `outtype` file and the necessary header and implementation files that must be included in a C++ application that runs against the object schema.

In summary, OCCI supports object handling in an Oracle database by:

- Improving application developer productivity by eliminating the requirement to write the host language variables that correspond to schema objects
- Maintaining SQL as the data definition language of choice by providing the ability to automatically map Oracle database schema objects created by SQL to host language variables; this allows Oracle to support a consistent, enterprise-wide model of the user's data
- Facilitating schema evolution of object types by regenerating included header files when the schema is changed, allowing Oracle applications to support schema evolution
- Executing SQL statements that manipulate object data and schema information
- Passing object references and instances as input variables in SQL statements
- Declaring object references and instances as variables to receive the output of SQL statements
- Fetching object references and instances from a database
- Describing properties of SQL statements that return object instances and references
Describing PL/SQL procedures or functions with object parameters or results

Extending commit and rollback calls to synchronize object and relational functionality

Advanced queuing of objects

OTT is typically invoked from the command line by specifying the intype file, the outtype file, and the specific database connection.

Additional Support

This section discusses how to build the OCCI examples that ship with Oracle Database, and additional resources.

Building OCCI Demos

Oracle ships with a directory of demos, parts of which are used as examples in this book. To build the examples, see the following steps:

1. Navigate to the demo directory.
   - On Windows, this directory is `ORACLE_HOME/rdbms/demo`.
   - On Linux and UNIX, this directory is `ORACLE_HOME/rdbms/demo`.

2. Ensure that this directory contains the file `demo_rdbms.mk`.

3. Create the OCCI demo using the `make` command.
   - To make all the demos at the same time, use the following parameters:
     ```shell
     make -f demo_rdbms.mk occidemos
     ```
   - To make only one demo, use parameters:
     ```shell
     make -f demo_rdbms.mk demoname
     ```
   - To make a single demo with objects, use parameters:
     ```shell
     make -f demo_rdbms.mk buildocci EXE=demoname OBJS=demoname.o
     ```
   - To make a single demo with static libraries, use parameters:
     ```shell
     make -f demo_rdbms.mk buildocci_static EXE=demoname OBJS=demoname.o
     ```
   - For more options, examine the `demo_rdbms.mk` file.

OCCI on the Oracle Technology Network

You can find additional information on OCCI, including a forum, downloads, and white papers, at:

This chapter provides an overview of installation and upgrading for Oracle C++ Call Interface (OCCI).

This chapter contains these topics:

- Installing Oracle C++ Call Interface
- Upgrading Considerations
- Determining the Oracle Database Versions
- Instant Client
- Instant Client Light (English)
- Using OCCI with Microsoft Visual C++

### Installing Oracle C++ Call Interface

OCCI is installed as part of the Oracle Database. To determine additional configuration requirements, you should refer to the Oracle Database Installation Guide and the Oracle Database Client Installation Guide that is specific to your platform.

### Upgrading Considerations

To use the new features available in this release, you must recompile and relink all OCCI applications, including classes generated through the Object Type Translator Utility, when upgrading from an earlier Oracle Client release.

### Determining the Oracle Database Versions

When an application uses several separate code paths with different server versions or client patchsets, you can verify these options both during compilation and at run time.

### Determining the Oracle Client Version During Compilation

The OCCI header files define OCCI_MAJOR_VERSION and OCCI_MINOR_VERSION macros. Example 2–1 illustrates one way to use these macros:

**Example 2–1  How to Determine the Major Client Version and Set Performance Features**

```c
#if (OCCI_MAJOR_VERSION > 9)
    env->setCacheSortedFlush(true); // benefit of performance, if available
#endif
```
Determining the Oracle Client and Server Versions at Run Time

During run time, you can check both the client and server versions of the current Connection by using the getClientVersion(), getServerVersion(), and getServerVersionUString() methods.

Instant Client

The Instant Client feature makes it extremely easy and fast to deploy OCCI based customer application by eliminating the need for ORACLE_HOME. The storage space requirements are an additional benefit; Instant Client shared libraries occupy about one-fourth of the disk space required for a full client installation.

Benefits of Instant Client

- Installation involves copying only four files.
- Storage space requirement for the client is minimal
- No loss of functionality or performance exists for deployed applications
- Simplified packaging with ISV applications

The OCCI Instant Client capability simplifies OCCI installation. Even though OCCI is independent of ORACLE_HOME setting in the Instant Client mode, applications that rely on ORACLE_HOME settings can continue operation by setting it to the appropriate value. The activation of the Instant Client mode is only dependent on the ability to load the Instant Client data shared library. In particular, this feature allows interoperability with Oracle applications that use ORACLE_HOME for their data, but use a newer release of Oracle Client.

Installing Instant Client

OCCI requires only four shared libraries (or dynamic link libraries, as they are called on some operating systems) to be loaded by the dynamic loader of the operating system. Oracle Database 11g Release 2 (11.2) library names are used; the number part of library names changes to remain consistent with future release numbers.

- OCI Instant Client Data Shared Library (libociei.so on Linux and UNIX and oraociei11.dll on Windows); correct installation of this file determines if you are operating in Instant Client mode
- Client Code Library (libclntsh.so.11.1 on Linux and UNIX and oci.dll on Windows)
- Security Library (libnnz11.so on Linux and UNIX and orannzsbb11.dll on Windows)
- OCCI Library (libocci.so.11.1 on Linux and UNIX and oraocci11.dll on Windows)

Oracle Technology Network

The Instant Client libraries are also available on the Oracle Technology Network (OTN) Web site at:
http://www.oracle.com/technology.tech/oci/instantclient/

If these four libraries are accessible through the directory on the Operating System Library Path variable (LD_LIBRARY_PATH on Linux and UNIX, and PATH on Windows), then OCCI operates in the Instant Client mode. In this mode, there is no
dependency on $ORACLE_HOME$ and none of the other code and data files provided in $ORACLE_HOME$ are needed by OCCI.

If you are installing Instant Client from the Oracle Technology Network,

1. Download and install the Instant Client libraries to an empty directory, such as instantclient_11_2.
2. Set the operating system shared library path environment variable (LD_LIBRARY_PATH on Linux and UNIX and PATH on Windows) to the directory used in step 1, instantclient_11_2.

**Instant Client SDK** Instant Client can also be downloaded as an SDK package. The SDK contains all necessary header files and a makefile for developing OCCI applications in an Instant Client environment. Once developed, these applications can be deployed in any client environment. The SDK has these additional features:

- It contains C++ demonstration programs.
- It includes libraries required to link applications on Windows, and a Make.bat file is provided to build demos.
- The Makefile demo.mk is provided to build the demos for Linux and UNIX. The instantclient_11_2 directory must be on the LD_LIBRARY_PATH before linking the application. These programs require symbolic links for the Client Code Library and the OCCI library, libclntsh.so and libocci.so respectively, in the instantclient_11_2 directory. The demo Makefile, demo.mk, generates these before the link step. These symbolic links can also be created in a shell script:

```bash
cd instantclient_11_2
ln -s libclntsh.so.11.1 libclntsh.so
ln -s libocci.so.11.1 libocci.so
```

- The SDK also contains the Object Type Translator (OTT) utility and its classes to generate the application header files.

**Complete Client Installation**
If you performed a complete client installation by choosing the Admin option,

- On Linux or UNIX platforms, the libociei.so library can be copied from the $ORACLE_HOME/instantclient directory. All the other libraries can be copied from the $ORACLE_HOME/lib directory in a full Oracle installation.
- On Windows, the oraociei11.dll library can be copied from the $ORACLE_HOME\instantclient directory. All other Windows libraries can be copied from the $ORACLE_HOME\bin directory. To use the Microsoft ODBC and OLEDB driver, ociw32.dll must also be copied from $ORACLE_HOME\bin.

**Oracle Universal Installer**
If you did not install the database, you can install these libraries by choosing the Instant Client option from the Oracle Universal Installer. After completing these steps, you can begin running OCCI applications.

1. Install the Instant Client shared libraries to a directory such as instantclient_11_2.
2. Set the operating system shared library path environment variable to the directory from step 1. For example, on Linux or UNIX, set the LD_LIBRARY_PATH to instantclient_11_2. On Windows, set PATH to locate the instantclient_11_2 directory.
Instant Client CD

You can also install Instant Client from the Instant Client CD. You must install Instant Client either in an empty directory or on a different system.

There should be only one set of Oracle libraries on the operating system Library Path variable; if you have several directories or copies of Instant Client libraries, only one directory should be on the operating system Library Path.

Similarly, if you also have an installation on an ORACLE_HOME of the same system, do not place both the ORACLE_HOME/lib and Instant Client directory on the operating system Library Path, regardless of the order in which they appear on the Library Path. Only one of ORACLE_HOME/lib directory (for non-Instant Client operation) or Instant Client directory (for Instant Client operation) should be on the operating system Library Path variable.

Using Instant Client

The Instant Client feature is designed for running production applications. For development, use either the Instant Client SDK or a full installation to access OCCI header files, makefiles, demonstration programs, and so on.

Patching Instant Client Shared Libraries on UNIX

This feature is not available on Windows platforms.

Because Instant Client is primarily a deployment feature, one of its design objectives is to reduce the number and size of necessary files. Therefore, Instant Client deployment does not include all files for patching shared libraries. You should use the OPATCH utility on an ORACLE_HOME-based full client to patch the Instant Client shared libraries. The OPATCH utility stores the patching information of the ORACLE_HOME installation in libclntsh.so.11.1 for Linux and UNIX. This information can be retrieved using the genezi utility:

genezi -v

If the genezi utility is not installed on the system that deploys Instant Client, you can copy it from the ORACLE_HOME/bin directory of the ORACLE_HOME system.

After applying the patch in an ORACLE_HOME environment, copy the files listed in "Installing Instant Client" on page 2-2 to the Instant Client directory.

Instead of copying individual files, you can generate Instant Client *.zip files, as described in "Regenerating the Data Shared Library and Zip Files". Then, instead of copying individual files, you can instead copy the zip files to the target system and unzip them.

Regenerating the Data Shared Library and Zip Files

This feature is not available on Windows platforms.

The Instant Client Data Shared Library, libociei.so, can be regenerated in a Client Admin Install of ORACLE_HOME. Executing Example 2–2 creates a new libociei.so file based on current file in ORACLE_HOME and place it in the ORACLE_HOME/rdbms/install/instantclient directory; the make target illibociei generates libociei.so.

This location of the regenerated data shared library, libociei.so, is different from the original location of ORACLE_HOME/instantclient

The script in Example 2–2 also creates a directory for Instant Client Light (English)
Example 2–2  How to Regenerate the Data Shared Library Files

mkdir -p $ORACLE_HOME/rdbms/install/instantclient/light

cd $ORACLE_HOME/rdbms/lib

make -f ins_rdbms.mk ilibociei

Database Connection Names for Instant Client

All Oracle net naming methods that do not require use of ORACLE_HOME or TNS_ADMIN to locate configuration files such as tnsnames.ora or sqlnet.ora work in the Instant Client mode.

The connectString parameter in the createConnection() call can be specified in the following formats:

- As an SQL Connect URL string, of the form:

  //host:[port]/[service name]

  such as:

  //myserver111:5521/bjava21

- As an Oracle Net keyword-value pair. For example:

  (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp) (HOST=myserver111) (PORT=5521))
  (CONNECT_DATA=(SERVICE_NAME=bjava21)))

- As a connection name that is resolved through Directory Naming when the site is configured for LDAP server discovery.

- As an entry in the tnsnames.ora file.

  If the TNS_ADMIN environment variable is not set, and TNSNAMES entries such as inst1 are used, then the ORACLE_HOME variable must be set and the configuration files are expected to be in the $ORACLE_HOME/network/admin directory.

  Naming methods that require TNS_ADMIN to locate configuration files continue to work if the TNS_ADMIN environment variable is set.

  The ORACLE_HOME variable in this case is only used for locating Oracle Net configuration files, and no other component of OCCI Client Code Library uses the value of ORACLE_HOME.

  The empty connectString parameter of createConnection() is supported by setting the environment variable (TWO_TASK on Linux and UNIX, and LOCAL on Windows) to one of the values described earlier.

  See Also: Oracle Database Net Services Administrator’s Guide for more information on the connect descriptor.

Environment Variables for OCCI Instant Client

The ORACLE_HOME environment variable no longer determines the location of Globalization Support, CORE, and error message files. An OCCI-only application should not require ORACLE_HOME to be set. However, if it is set, it does not have an impact on OCCI’s operation. OCCI always obtains its data from the Data Shared Library. If the Data Shared Library is not available, only then is ORACLE_HOME used and a full client installation is assumed. When set, ORACLE_HOME should be a valid operating system path name that identifies a directory.
Environment variables ORA_NLS33, ORA_NLS32, and ORA-NLS are ignored in the Instant Client mode.

In the Instant Client mode, if the ORA_TZFILE variable is not set, then the larger, default, timezone.n.dat file (where n is the version number of the file) from the Data Shared Library is used. If using the smaller timezone.n.dat file from the Data Shared Library, then set the ORA_TZFILE environment variable to the name of the file without any absolute or relative path names, as shown in Example 2–3.

Example 2–3  How to set the ORA_TZFILE Environment Variable

On Linux and UNIX:
setenv ORA_TZFILE timezone_n.dat

On Windows:
set ORA_TZFILE timezone_n.dat

If OCCI is not operating in the Instant Client mode because the Data Shared Library is not available, the ORA_TZFILE variable, if set, names a complete path name.

If TNSNAMES entries are used, then TNS_ADMIN directory must contain the TNSNAMES configuration files. If TNS_ADMIN is not set, the ORACLE_HOME/network/admin directory must contain Oracle Net Services configuration files.

Instant Client Light (English)

Instant Client Light (English) further reduces installation space requirements of the client installation over Instant Client by another 63 MB. Specifically, the installation of the Instant Client Light (English) shared library, libociicus.so on Linux and UNIX and oraociicus11.dll for Windows, occupies 4 MB on UNIX platforms, when the full Instant Client shared library, libociie.so, occupies 67 MB of disk space.

Instant Client Light (English), as the name implies, is geared toward applications that require English-only error messages and use either US7ASCII, WE8DEC, or a Unicode characterset. Instant Client Light (English) also has no restrictions on the TERRITORY field of the NLS_LANG setting. As a result, applications that meet these characterset and territory criteria can significantly reduce its footprint if they operate in the Instant Client Light (English) environment.

Globalization Settings for Instant Client Light (English)

Instant Client Light (English) supports the following character sets:

- Single-byte character sets include US7ASCII, WE8DEC, WE8MSWIN1252, and WE8ISO8859P1.
- Unicode character sets include UTF8, AL16UTF16, and AL32UTF8.

Instant Client Light (English) returns an error message if the application attempts to use a character set or a national character set not listed here, either on the client or on the database. The possible error messages, listed here, are only available in English:

- ORA-12734 Instant Client Light: unsupported client national character set (NLS_LANG value set)
- ORA-12735 Instant Client Light: unsupported client character set (NLS_LANG value set)
■ ORA-12736 Instant Client Light: unsupported server national character set (NLS_LANG value set)
■ ORA-12737 Instant Client Light: unsupported server character set (NLS_LANG value set)

When setting NLS_LANG parameters, use the following:

```
American_territory.charset
```

where territory is any valid Territory that can be specified through NLS_LANG, and charset is a character set listed in this section.

**See Also:** Oracle Database Globalization Support Guide for more information about NLS settings.

Using Instant Client Light (English)

To determine whether to operate in the Instant Client mode, OCCI applications look for the Data Shared Library on the LD_LIBRARY_PATH for Linux and UNIX and PATH on Windows. If this library is not found, OCCI attempts to load the Instant Client Light (English) Data Shared Library, libociicus.so for Linux and UNIX and oraociicus11.dll on Windows. If neither is found, a full ORACLE_HOME installation is assumed.

Installing Instant Client Light (English)

There are three ways to install Instant Client Light (English): from Oracle Technology Network Download, through Client Admin Install, and through Oracle Universal Installer.

Note that all Instant Client and Instant Client Light (English) files should always be copied or installed into an empty directory to ensure that there are no incompatible binaries in the final installation.

**Oracle Technology Network Download**

When installing Instant Client Light (English) from Oracle Technology Network (OTN), download and unzip the basiclite.zip package instead of the usual basic.zip package. You must ensure that the instantclient_11_2 directory is empty before unzipping the libraries. The downloadable package is at the following URL on OTN:

```
http://www.oracle.com/technology/tech/oci/instantclient/
```

**Client Admin Install**

Instead of copying the Instant Client Data Shared Library from the ORACLE_HOME/instantclient directory, use the Instant Client Light (English) Data Shared Library, libociicus.so for Linux and UNIX and oraociicus11.dll for Windows, from the ORACLE_HOME/instantclient/light directory. In other words, the Instant Client directory on the LD_LIBRARY_PATH for Linux and UNIX and PATH for Windows should contain the smaller Instant Client Light (English) Data Shared Libraries.

**Oracle Universal Installer**

If the Instant Client option is selected from the Oracle Universal Installer (OUI), the full Instant Client is installed by default, but the libraries for Instant Client Light (English) are also installed. To operate in Instant Client Light (English) mode, the
Instant Client Light (English) Data Shared Library must replace the Instant Client library. Therefore, you must place libociicus.so on the LD_LIBRARY_PATH for Linux and UNIX, and oraociicus11.dll on the PATH for Windows. This design ensures that the Instant Client Light (English) is not enabled by default.

The Instant Client Light (English) Data Shared Library is initially placed in the ORACLE_HOME/instantclient/light directory. You must move it to the base directory of the installation, ORACLE_HOME/instantclient, and remove the Instant Client Data Shared Library in that directory.

**Example 2–4 Installing Instant Client Light (English) through Oracle Universal Installer**

If the OUI has installed the Instant Client in my_oraic_11_1 directory on the LD_LIBRARY_PATH, then the following commands would ensure operation in the Instant Client Light (English) mode. Note that to avoid use of incompatible binary files, all Instant Client files should be copied and installed in an empty directory.

```
cd my_oraic_11_1
rm libociei.so
mv light/libociicus.so .
```

**Using OCCI with Microsoft Visual C++**

The Oracle Database 11gR1 release includes OCCI libraries for developing applications with Microsoft Visual C++ version 8.0 (.NET 2005) Microsoft Visual C++ version 7.1 (.NET 2003). Microsoft Visual C++ version 7.0 is no longer supported.

Microsoft Visual C++ version 7.1 libraries are installed in the following default locations:

```
ORACLE_BASE\ORACLE_HOME\bin\oraocci11.dll
ORACLE_BASE\ORACLE_HOME\oci\lib\msvc\oraocci11.lib
```

Copies of these two files are also installed under the directory:

```
ORACLE_BASE\ORACLE_HOME\oci\lib\msvc\vc71
```

The Microsoft Visual C++ 8.0-specific version of the libraries is installed under:

```
ORACLE_BASE\ORACLE_HOME\oci\lib\msvc\vc8
```

Applications should link with the appropriate OCCI library. You must ensure that the corresponding DLL is located in the Windows system PATH.

Applications that link to MSVCRTD.DLL, a debug version of Microsoft C-Runtime, /MDd compiler flag, should link with these specific OCCI libraries: oraocci11d.lib and oraocci11d.dll.

All Instant Client packages contain the version of the OCCI DLL that is compatible with Microsoft Visual C++ version 7.1.
This chapter describes the basics of developing C++ applications using Oracle C++ Call Interface (OCCI) to work with data stored in relational databases.

This chapter contains these topics:

- Connecting to a Database
- Pooling Connections
- Executing SQL DDL and DML Statements
- Types of SQL Statements in the OCCI Environment
- Executing SQL Queries
- Executing Statements Dynamically
- Committing a Transaction
- Caching Statements
- Handling Exceptions

## Connecting to a Database

You have several different options for how your application connects to the database.

## Creating and Terminating an Environment

All OCCI processing takes place inside the `Environment` class. An OCCI environment provides application modes and user-specified memory management functions. Example 3–1 illustrates how to create an OCCI environment.

### Example 3–1  How to Create an OCCI Environment

```cpp
Environment *env = Environment::createEnvironment();
```

All OCCI objects created with the `createxxx()` methods (connections, connection pools, statements) must be explicitly terminated. When appropriate, you must also explicitly terminate the environment. Example 3–2 shows how you terminate an OCCI environment.

### Example 3–2  How to Terminate an OCCI Environment

```cpp
Environment::terminateEnvironment(env);
```
In addition, an OCCI environment should have a scope that is larger than the scope of the following object types created inside that environment: Agent, Bytes, Date, Message, IntervalDS, IntervalYM, Subscription, and Timestamp. This rule does not apply to BFile, Blob, and Clob objects, as demonstrated in Example 3–3.

**Example 3–3  How to Use Environment Scope with Blob Objects**

```cpp
const string userName = "HR";
const string password = "password";
const string connectString = ""

Environment *env = Environment::createEnvironment();
{
    Connection *conn = env->createConnection(
        userName, password, connectString);
    Statement *stmt = conn->createStatement(
        "SELECT blobcol FROM mytable");
    ResultSet *rs = stmt->executeQuery();
    rs->next();
    Blob b = rs->getBlob(1);
    cout << "Length of BLOB : " << b.length();
    ...
    stmt->closeResultSet(rs);
    conn->terminateStatement(stmt);
    env->terminateConnection(conn);
}
Environment::terminateEnvironment(env);
```

If the application requires access to objects in the global scope, such as static or global variables, these objects must be set to NULL before the environment is terminated. In the preceding example, if `b` was a global variable, a `b.setNull()` call has to be made before the `terminateEnvironment()` call.

You can use the mode parameter of the `createEnvironment()` method to specify that your application:

- Runs in a threaded environment (`THREADED_MUTEXED` or `THREADED_UNMUTEXED`)
- Uses objects (`OBJECT`)

The mode can be set independently in each environment.

### Opening and Closing a Connection

The `Environment` class is the factory class for creating `Connection` objects. You first create an `Environment` instance, and then use it to enable users to connect to the database through the `createConnection()` method.

**Example 3–4  How to Create an Environment and then a Connection to the Database**

```cpp
Environment *env = Environment::createEnvironment();
Connection *conn = env->createConnection("HR", "password");
```

You must use the `terminateConnection()` method shown in the following code example to explicitly close the connection at the end of the working session. In addition, the OCCI environment should be explicitly terminated.
You should remember that all objects (Refs, Bfiles, Producers, Consumers, and so on) created or named within a Connection instance must be within the inner scope of that instance; the scope of these objects must be explicitly terminated before the Connection is terminated. Example 3–5 demonstrates how to terminate the connection and the environment.

Example 3–5  How to Terminate a Connection to the Database and the Environment

```cpp
env->terminateConnection(conn);
Environment::terminateEnvironment(env);
```

Pooling Connections

This section discusses how to use the connection pooling feature of OCCI. The information covered includes the following topics:

- Using Connection Pools
- Stateless Connection Pooling

The primary difference between the two is that StatelessConnectionPools are used for applications that do not depend on state considerations; these applications can benefit from performance improvements available through use of pre-authenticated connections.

Using Connection Pools

For many middle-tier applications, connections to the database should be enabled for a large number of threads. Because each thread exists for a relatively short time, opening a connection to the database for every thread would be inefficient use of connections, and would result in poor performance.

By employing the connection pooling feature, your application can create a small set of connections available to a large number of threads, enabling you to use database resources very efficiently.

Creating a Connection Pool

To create a connection pool, you use the createConnectionPool() method on page 13-87, as demonstrated in Example 3–6.

Example 3–6  The createConnectionPool() Method

```cpp
virtual ConnectionPool* createConnectionPool(
    const string &poolUserName,
    const string &poolPassword,
    const string &connectString = "",
    unsigned int minConn = 0,
    unsigned int maxConn = 1,
    unsigned int incrConn = 1) = 0;
```

The following parameters are used in Example 3–6:

- poolUserName: The owner of the connection pool
- poolPassword: The password to gain access to the connection pool
- connectString: The database name that specifies the database server to which the connection pool is related
Pooling Connections

- **minConn**: The minimum number of connections to be opened when the connection pool is created
- **maxConn**: The maximum number of connections that can be maintained by the connection pool. When the maximum number of connections are open in the connection pool, and all the connections are busy, an OCCI method call that needs a connection waits until it gets one, unless setErrorOnBusy() was called on the connection pool.
- **incrConn**: The additional number of connections to be opened when all the connections are busy and a call needs a connection. This increment is implemented only when the total number of open connections is less than the maximum number of connections that can be opened in that connection pool.

**Example 3–7** demonstrates how you can create a connection pool.

**Example 3–7  How to Create a Connection Pool**

```cpp
const string connectString = "";
unsigned int maxConn = 5;
unsigned int minConn = 3;
unsigned int incrConn = 2;

ConnectionPool *connPool = env->createConnectionPool(
    poolUserName,
    poolPassword,
    connectString,
    minConn,
    maxConn,
    incrConn);
```

You can also configure all these attributes dynamically. This lets you design an application that has the flexibility of reading the current load (number of open connections and number of busy connections) and tune these attributes appropriately. In addition, you can use the `setTimeOut()` method to time out the connections that are idle for more than the specified time. The OCCI terminates idle connections periodically to maintain an optimum number of open connections.

There is no restriction that one environment must have only one connection pool. There can be multiple connection pools in a single OCCI environment, and these can connect to the same or different databases. This is useful for applications requiring load balancing.

**Proxy Connections**

If you authorize the connection pool user to act as a proxy for other connections, then no password is required to log in database users who use connections in the connection pool.

A proxy connection can be created by using either of the following two versions of the `createProxyConnection()` method on page 13-63, demonstrated in **Example 3–8**.

**Example 3–8  The createProxyConnection() Method**

```cpp
ConnectionPool->createProxyConnection(
    const string &username,
    Connection::ProxyType proxyType = Connection::PROXY_DEFAULT);
```

or

```
ConnectionPool->createProxyConnection(
    Connection::ProxyType proxyType = Connection::PROXY_DEFAULT);
```
const string &username,
string roles[],
int numRoles,
Connection::ProxyType proxyType = Connection::PROXY_DEFAULT);

The following parameters are used in the previous method example:

- roles[]: The roles array specifies a list of roles to be activated after the proxy connection is activated for the client.
- Connection::ProxyType proxyType = Connection::PROXY_DEFAULT: The enumeration Connection::ProxyType lists constants representing the various ways of achieving proxy authentication. PROXY_DEFAULT is used to indicate that name represents a database username and is the only proxy authentication mode currently supported.

**Stateless Connection Pooling**

Stateless Connection Pooling is specifically designed for use in applications that require short connection times and do not deal with state considerations. The primary benefit of Stateless Connection Pooling is increased performance, since the time consuming connection and authentication protocols are eliminated.

Stateless Connection Pools create and maintain a group of stateless, authenticated connection to the database that can be used by multiple threads. Once a thread finishes using its connection, it should release the connection back to the pool. If no connections are available, new ones are generated. Thus, the number of connections in the pool can increase dynamically.

Some connections in the pool may be tagged with specific properties. The user may request a default connection, set certain attributes, such as Globalization Support settings, then tag it and return it to the pool. When a connection with same attributes is needed, a request for a connection with the same tag can be made, and one of several connections in the pool with the same tag can be reused. The tag on a connection can be changed or reset.

Proxy connections may also be created and maintained through the Stateless Connection Pooling interface.

Stateless connection pooling improves the scalability of the mid-tier applications by multiplexing the connections. However, connections from a StatelessConnectionPool should not be used for long transactions, as holding connections for long periods leads to reduced concurrency.

---

**Caution:**

- OCCI does not check for the correctness of the connection-tag pair. You are responsible for ensuring that connections with different client-side properties do not have the same tag.
- Your application should commit or rollback any open transactions before releasing the connection back to the pool. If this is not done, Oracle automatically commits any open transactions when the connection is released.

---

There are two types of stateless connection pools:

- A **homogeneous pool** is one in which all the connections are authenticated with the username and password provided at the time of creation of the pool.
Therefore, all connections have the same authentication context. Proxy connections are not allowed in such pools.

- Different connections can be authenticated by different usernames in heterogeneous pools. Proxy connections can also exist in heterogeneous pools, provided the necessary privileges for creating them are granted on the server. Additionally, heterogeneous pools support external authentication.

Example 3–9 illustrates a basic usage scenario for connection pools. Example 3–10 presents the usage scenario for creating and using a homogeneous stateless connection pool, while Example 3–11 covers the use of heterogeneous pools.

**Example 3–9  How to Use a StatelessConnectionPool**

Because the pool size is dynamic, in response to changing user requirements, up to the specified maximum number of connections. Assume that a stateless connection pool is created with the following parameters:

- minConn = 5
- incrConn = 2
- maxConn = 10

Five connections are opened when the pool is created:

- openConn = 5

Using get[AnyTagged][Proxy]Connection() methods, the user consumes all 5 open connections:

- openConn = 5
- busyConn = 5

When the user wants another connection, the pool opens 2 new connections and returns one of them to the user.

- openConn = 7
- busyConn = 6

The upper limit for the number of connections that can be pooled is maxConn specified at the time of creation of the pool.

The user can also modify the pool parameters after the pool is created using the call to setPoolSize() method.

If a heterogeneous pool is created, the incrConn and minConn arguments are ignored.

**Example 3–10  How to Create and Use a Homogeneous Stateless Connection Pool**

To create a homogeneous stateless connection pool, follow these basic steps and pseudocode commands:

1. Create a stateless connection pool in the HOMOGENEOUS mode of the Environment with a createStatelessConnectionPool() call.

```cpp
StatelessConnectionPool *scp =
    env->createStatelessConnectionPool(
        username, passwd, connectString, maxCon, minCon, incrCon,
        StatelessConnectionPool::HOMOGENEOUS );
```
2. Get a new or existing connection from the pool by calling the `getConnection()` method.

   ```
   Connection *conn=scp->getConnection(tag);
   ```

   During the execution of this call, the pool is searched for a connection with a matching tag. If such a connection exists, it is returned to the user. Otherwise, an untagged connection authenticated by the pool username and password is returned.

   Alternatively, you can obtain a connection with `getAnyTaggedConnection()` call. It returns a connection with a non-matching tag if neither a matching tag or NULL tag connections are available. You should verify the tag returned by a `getTag()` call on `Connection`.

   ```
   Connection *conn=scp->getAnyTaggedConnection(tag);
   string tag=conn->getTag();
   ```

3. Use the connection.

4. Release the connection to the `StatelessConnectionPool` through the `releaseConnection()` call.

   ```
   scp->releaseConnection(conn, tag);
   ```

   An empty tag, "", untags the `Connection`.

   You have an option of retrieving the connection from the `StatelessConnectionPool` using the same `tag` parameter value in a `getConnection()` call.

   ```
   Connection *conn=scp->getConnection(tag);
   ```

   Instead of returning the `Connection` to the `StatelessConnectionPool`, you may want to destroy it using the `terminateConnection()` call.

   ```
   scp->terminateConnection(conn);
   ```

5. Destroy the pool through a `terminateStatelessConnectionPool()` call on the `Environment` object.

   ```
   env->terminateStatelessConnectionPool(scp);
   ```

**Example 3–11 How to Create and Use a Heterogeneous Stateless Connection Pool**

To create a heterogeneous stateless connection pool, follow these basic steps and pseudocode commands:

1. Create a stateless connection pool in the `HETEROGENEOUS` mode of the `Environment` with a `createStatelessConnectionPool()` call.

   ```
   StatelessConnectionPool *scp =
   env->createStatelessConnectionPool(
       username, passwd, connectionString, maxCon, minCon, incrCon,
       StatelessConnectionPool::HETEROGENEOUS);
   ```

   If you are enabling external authentication, you must also activate the `USES_EXT_AUTH` mode in the `createStatelessConnectionPool()` call.

   ```
   StatelessConnectionPool *scp =
   env->createStatelessConnectionPool(
       username, passwd, connectionString, maxCon, minCon, incrCon,
       StatelessConnectionPool::PoolType{Relational Programming  3-7
   ```
StatelessConnectionPool::USES_EXT_AUTH |
StatelessConnectionPool::HETEROGENEOUS));

2. Get a new or existing connection from the pool by calling the getConnection() method of the StatelessConnectionPool that is overloaded for the heterogeneous pool option.

Connection *conn=scp->getConnection(username, passwd, tag);

During the execution of this call, the heterogeneous pool is searched for a connection with a matching tag. If such a connection exists, it is returned to the user. Otherwise, an appropriately authenticated untagged connection with a NULL tag is returned.

Alternatively, you can obtain a connection with getAnyTaggedConnection() call that has been overloaded for heterogeneous pools. It returns a connection with a non-matching tag if neither a matching tag or NULL tag connections are available. You should verify the tag returned by a getTag() call on Connection.

Connection *conn=scp->getAnyTaggedConnection(username, passwd, tag);
string tag=conn->getTag();

You may also want to use proxy connections by getProxyConnection() or getAnyTaggedProxyConnection() calls on the StatelessConnectionPool.

Connection *pconn = scp->getProxyConnection(proxyName, roles{},
nuRoles, tag, proxyType);
Connection *pconn = scp->getAnyTaggedProxyConnection( proxyName, tag,
proxyType);

If the pool supports external authentication, use the following getConnection() call:
Connection *conn=scp->getConnection();

3. Use the connection.

4. Release the connection to the StatelessConnectionPool through the releaseConnection() call.

scp->releaseConnection(conn, tag);

An empty tag, " ", untags the Connection.

You have an option of retrieving the connection from the StatelessConnectionPool using the same tag parameter value in a getConnection() call.
Connection *conn=scp->getConnection(tag);

Instead of returning the Connection to the StatelessConnectionPool, you may want to destroy it using the terminateConnection() call.
scp->terminateConnection(conn);

5. Destroy the pool through a terminateStatelessConnectionPool() call on the Environment object.
env->terminateStatelessConnectionPool(scp);
Database Resident Connection Pooling

Enterprise-level applications must typically handle a high volume of simultaneous user sessions that are implemented as persistent connections to the database. The memory overhead of creating and managing these connections has significant implications for the performance of the database.

Database Resident Connection Pooling solves the problem of too many persistent connections by providing a pool of dedicated servers for handling a large set of application connections, thus enabling the database to scale to tens of thousands of simultaneous connections. It significantly reduces the memory footprint on the database tier and increases the scalability of both the database and middle tiers. Database Resident Connection Pooling is designed for architectures with multi-process application servers and multiple middle tiers that cannot accommodate connection pooling in the middle tier.

Database Resident Connection Pooling architecture closely follows the default dedicated model for connecting to an Oracle Database instance; however, it removes the overhead of assigning a specific server to each connection. On the server tier, most connections are inactive at any given time, and each of these connections consumes memory. Therefore, database systems that support high connection volumes face the risk of quickly exhausting all available memory. Database Resident Connection Pooling allows a connection to use a dedicated server, which combines an Oracle server process and a user session. Once the connection becomes inactive, it returns its resources to the pool, for use by similar connections.

In multithreaded middle tiers that are capable of comprehensive connection pooling, the issue of unused connections is somewhat different. As the number of middle tiers increases, each middle tier privately holds several connections to the database; these connections cannot be shared with other middle tiers. Locating the connection pool on the database instead enables the sharing of connections across similar clients.

Database Resident Connection Pooling supports password-based authentication, statement caching, tagging, and Fast Application Notification. You can also use client-side stateless connection pooling with the database resident connection pooling. Note that clients that hold connections from the database resident connection pool are persistently connected to a background Connection Broker process. The Connection Broker implements the pool functionality and multiplexes inbound client connections to a pool of dedicated server processes. Clients that do not use the connection pool use dedicated server processes instead.

See Also:

- Oracle Database Concepts for details about the architecture of Database Resident Connection Pooling
- Oracle Database Administrator’s Guide for details on configuring Database Resident Connection Pooling
- Oracle Database PL/SQL Packages and Types Reference, for the DBMS_CONNECTION_POOL package

Administrating Database Resident Connection Pools

To implement database resident connection pooling, it must first be enabled on the system by a user with SYSDBA privileges. See Example 3–12 for steps necessary to initiate and maintain a database resident connection pool.
Example 3–12  How to Administer the Database Resident Connection Pools

A user with SYSDBA privileges must perform the next steps.

1. Connect to the database.
   SQLPLUS / AS SYSDBA

2. [Optional] Configure the parameters of the database resident connection pool. The default values of a pool are set in the following way:
   
   DBMS_CONNECTION_POOL.CONFIGURE_POOL("SYS_DEFAULT_CONNECTION_POOL",
   MIN=>10,
   MAX=>200);

3. [Optional] Alter specific parameters of the database resident connection pool without affecting other parameters.
   
   DBMS_CONNECTION_POOL.ALTER_PARAM("SYS_DEFAULT_CONNECTION_POOL",
   'INACTIVITY_TIMEOUT',
   10);

4. Start the connection pool. After this step, the connection pool is available to all qualified clients.
   
   DBMS_CONNECTION_POOL.START_POOL("SYS_DEFAULT_CONNECTION_POOL");

5. [Optional] Change the parameters of the database resident connection pool.
   
   DBMS_CONNECTION_POOL.ALTER_PARAM("SYS_DEFAULT_CONNECTION_POOL",
   'MAXSIZE',
   20);

6. [Optional] The configuration of the connection pool can be reset to default values.
   
   DBMS_CONNECTION_POOL.RESTORE_DEFAULTS("SYS_DEFAULT_CONNECTION_POOL");

7. Stop the pool. Note that pool information is persistent: stopping the pool does not destroy the pool name and configuration parameters.
   
   DBMS_CONNECTION_POOL.STOP_POOL();

Note that in Oracle RAC configurations, the database resident connection pool starts on all configured nodes. If the pool is not stopped, the starting configuration is persistent across instance restarts: the pool is started automatically when the instance comes up.

Using Database Resident Connection Pools

To use database resident connection pooling, you must specify the connection class and connection purity. If the application requests a connection that cannot be potentially tainted with prior connection state, it must specify purity as NEW; Oracle recommends this approach if clients from different geographic locale settings share the same database instance. When the application can use a previously used connection, the purity should be set to SELF. In conjunction with connection class and purity specifications, you can also use an application-specific tag to choose a previously used connection that has the desired state. The default connection pool name, as demonstrated in Example 3–12, is SYS_DEFAULT_CONNECTION_POOL.

This feature overloads StatelessConnectionPool Class and Environment Class interfaces for retrieving a connection (getConnection() and getProxyConnection()) by adding the parameters that specify connection class and purity. Every connection request outside of a client-side connection pool has a
default purity of NEW. Connection requests inside a client-side connection pool have a
default purity of SELF.

**Example 3–13  How to Get a Connection from a Database Resident Connection Pool**

```cpp
conn1 = env->createConnection /*username */"hr",
/*password*/ "password", /* database*/ "inst1_cmon",
/* connection class */"TESTCC", /* purity */Connection::SELF);
stmt1 = conn1->createStatement("select count(*) from emp");
rs=stmt1->executeQuery();
while ([rs->next()])
{
    int num = rs->getInt(1);
    sprintf((char *)tmp, "%d", num);
    cout << tmp << endl;
}
stmt1->closeResultSet(rs);
env->terminateConnection(conn1);
```

**Example 3–14 Using Client-Side Pool and Server-Side Pool**

```cpp
StatelessConnectionPool *scPool;
COCIConnection *conn1, *conn2;
scPool = env->createStatelessConnectionPool
    (poolUserName, poolPassword, connectString, maxConn,
    minConn, incrConn, StatelessConnectionPool::HOMOGENEOUS);
conn1= scPool->getConnection(/* Connection class name */"TESTCC",
/* Purity */ Connection::SELF);
/* or, for proxy connections */
conn2= scPool->getProxyConnection(/* username*/ "HR_PROXY",
/*Connection class */"TESTCC", /* Purity */Connection::SELF);
/* or, for getting a tagged connection */
conn3 = scPool->getConnection(/*connection class */"TESTCC",
/*purity*/ Connection::SELF,
/*tag*/ "TESTTAG");
/* Releasing a tagged connection is done presently */
scPool->releaseConnection(conn3, "TESTTAG");
/* To specify purity as new */
conn4 = scPool->getConnection(/* connection class */"TESTCC","/purity of new */
Connection::NEW);
/* Get a connection using username and password */
conn5 = scPool->getConnection (username, password,"TESTCC", Connection::SELF);
/* Using roles when asking for a connection */
conn6 = scPool->getProxyConnection (username, roles, nRoles,"TESTCC",
Connection::SELF);
...
/* The other code continues as is...writing for clarity */
...
stmt1=conn1->createStatement /*INSERT INTO emp values (:c1, :c2");
stmt1->setInt(1, thrid);
stmt1->setString(2, "Test");
int count = stmt1->executeUpdate ();
conn1->commit();
```
Executing SQL DDL and DML Statements

SQL is the industry-wide language for working with relational databases. In OCCI you execute SQL commands through the `Statement` class.

Creating a Statement Object

To create a `Statement` object, call the `createStatement()` method of the `Connection` object, as demonstrated in Example 3-15:

```
Example 3-15  How to Create a Statement
Statement *stmt = conn->createStatement();
```

Creating a Statement Object that Executes SQL Commands

Once you have created a `Statement` object, execute SQL commands by calling the `execute()`, `executeUpdate()`, `executeArrayUpdate()`, or `executeQuery()` methods on the `Statement` object. These methods are used for the following purposes:

- `execute()` executes all nonspecific statement types
- `executeUpdate()` executes DML and DDL statements
- `executeArrayUpdate()` executes multiple DML statements
- `executeQuery()` executes a query

Creating a Database Table

Example 3-16 demonstrates how you can create a database table using the `executeUpdate()` method.

```
Example 3-16  How to Create a Database Table Using the executeUpdate() Method
stmt->executeUpdate("CREATE TABLE shopping_basket
(item_number VARCHAR2(30), quantity NUMBER(3))");
```

Inserting Values into a Database Table

Similarly, you can execute a SQL INSERT statement by invoking the `executeUpdate()` method, as demonstrated in Example 3-17.

```
Example 3-17  How to Add Records Using the executeUpdate() Method
stmt->executeUpdate("INSERT INTO shopping_basket
VALUES('MANGO', 3)");
```

The `executeUpdate()` method returns the number of rows affected by the SQL statement.

See Also: `$ORACLE_HOME/rdbms/demo` for a code example that demonstrates how to perform insert, select, update, and delete operations on table rows.
Reusing the Statement Object

You can reuse a Statement object to execute SQL statements multiple times. To repeatedly execute the same statement with different parameters, you should specify the statement by the setSQL() method of the Statement object on page 13-244, as demonstrated in Example 3–18.

Example 3–18  How to Specify a SQL Statement Using the setSQL() Method

stmt->setSQL("INSERT INTO shopping_basket VALUES(:1,:2)");

You may now execute this INSERT statement as many times as required. If at a later time you want to execute a different SQL statement, you simply reset the statement object, as demonstrated in Example 3–19.

Example 3–19  How to Reset a SQL Statement Using the setSQL() Method

stmt->setSQL("SELECT * FROM shopping_basket WHERE quantity >= :1");

By using the setSQL() method on page 13-244, OCCI statement objects and their associated resources are not allocated or freed unnecessarily. To retrieve the contents of the current statement object at any time, use the getSQL() method on page 13-226.

Terminating a Statement Object

You should explicitly terminate and deallocate a Statement object using the terminateStatement() method on page 13-59, as demonstrated in Example 3–20.

Example 3–20  How to Terminate a Statement Using the terminateStatement() Method

Connection::conn->terminateStatement(Statement *stmt);

Types of SQL Statements in the OCCI Environment

There are three types of SQL statements in the OCCI environment:

- **Standard Statements** use SQL commands with specified values
- **Parameterized Statements** have parameters, or bind variables
- **Callable Statements** call stored PL/SQL procedures and functions

The methods of the Statement Class are subdivided into those applicable to all statements, to parameterized statements, and to callable statements. Standard statements are a superset of parameterized statements, and parameterized statements are a superset of callable statements.

Standard Statements

Both Example 3–16 and Example 3–17 demonstrate standard statements in which you must explicitly define the values of the statement. In Example 3–16, the CREATE TABLE statement specifies the name of the table shopping_basket. In Example 3–17, the INSERT statement stipulates the values that are inserted into the table, ('MANGO', 3).

Parameterized Statements

You can execute the same statement with different parameters by setting placeholders for the input variables of the statement. These statements are referred to as
parameterized statements because they can accept parameter input from a user or a program.

If you want to execute an INSERT statement with different parameters, you must first specify the statement by the setSQL() method of the Statement object on page 13-244, as demonstrated in Example 3–18.

You then call the setxxx() methods to specify the parameters, where xxx stands for the type of the parameter. Provided that the value of the statement object is "INSERT INTO shopping_basket VALUES(:1,:2)", as specified in Example 3–18, you can use the code in Example 3–21 to invoke the setString() method on page 13-245 and setInt() method on page 13-240 to input the values of these types into the first and second parameters, and the executeUpdate() method on page 13-219 to insert the new row into the table. You can reuse the statement object by re-setting the parameters and again calling the executeUpdate() method on page 13-219. If your application is executing the same statement repeatedly, you should avoid changing the input parameter types because this initiates a rebind operation, and affects application performance.

Example 3–21  How to Use setxxx() Methods to Set Individual Column Values
stmt->setString(1, "Banana");     // value for first parameter
stmt->setInt(2, 5);               // value for second parameter
stmt->executeUpdate();            // execute statement
...
stmt->setString(1, "Apple");      // value for first parameter
stmt->setInt(2, 9);               // value for second parameter
stmt->executeUpdate();            // execute statement

Callable Statements

PL/SQL stored procedures, as their name suggests, are procedures that are stored on the database server for reuse by an application. In OCCI, a callable statement is a call to a procedure which contains other SQL statements.

If you want to call a procedure countGroceries(), that returns the quantity of a specified kind of fruit, you must first specify the input parameters of a PL/SQL stored procedure through the setXXX() methods of the Statement class, as demonstrated in Example 3–22.

Example 3–22  How to Specify the IN Parameters of a PL/SQL Stored Procedure
stmt->setSQL(“BEGIN countGroceries(:1, :2); END:”);
int quantity;
stmt->setString(1, "Apple");       // specify the first (IN) parameter of procedure

However, before calling a stored procedure, you must specify the type and size of any OUT parameters by calling the registerOutParam() method on page 13-231, as demonstrated in Example 3–23. For IN/OUT parameters, use the setXXX() methods to pass in the parameter, and getXXX() methods to retrieve the results.

Example 3–23  How to Specify OUT Parameters of a PL/SQL Stored Procedure
stmt->registerOutParam(2, Type::OCCIINT, sizeof(quantity));
   // specify type and size of the second (OUT) parameter

You now execute the statement by calling the procedure:
stmt->executeUpdate();            // call the procedure
Finally, you obtain the output parameters by calling the relevant `getxxx()` method:

```c
quantity = stmt->getInt(2);    // get value of the second (OUT) parameter
```

### Callable Statements that Use Array Parameters

A PL/SQL stored procedure executed through a callable statement can have array of values as parameters. The number of elements in the array and the dimension of elements in the array are specified through the `setDataBufferArray()` method.

The following example shows the `setDataBufferArray()` method:

```c
void setDataBufferArray(
    unsigned int paramIndex,
    void *buffer,
    Type type,
    ub4 arraySize,
    ub4 *arrayLength,
    sb4 elementSize,
    ub2 *elementLength,
    sb2 *ind = NULL,
    ub2 *rc = NULL);
```

The following parameters are used in the previous method example:

- **paramIndex**: Parameter number
- **buffer**: Data buffer containing an array of values
- **Type**: Type of data in the data buffer
- **arraySize**: Maximum number of elements in the array
- **arrayLength**: Number of elements in the array
- **elementSize**: Size of the current element in the array
- **elementLength**: Pointer to an array of lengths. `elementLength[i]` has the current length of the `i`th element of the array
- **ind**: Indicator information
- **rc**: Returns code

### Streamed Reads and Writes

OCCI supports a streaming interface for insertion and retrieval of very large columns by breaking the data into a series of small chunks. This approach minimizes client-side memory requirements. This streaming interface can be used with parameterized statements such as `SELECT` and various DML commands, and with callable statements in PL/SQL blocks. The data types supported by streams are `BLOB`, `CLOB`, `LONG`, `LONG RAW`, `RAW`, and `VARCHAR2`.

Streamed data is of three kinds:

- **A writable** stream corresponds to a bind variable in a `SELECT/DML` statement or an `IN` argument in a callable statement.
- **A readable** stream corresponds to a fetched column value in a `SELECT` statement or an `OUT` argument in a callable statement.
- **A bidirectional** stream corresponds to an `IN/OUT` bind variable.

Methods of the **Stream Class** support the stream interface.
The `getStream()` method of the `Statement` class returns a stream object that supports reading and writing for DML and callable statements:

- For writing, it passes data to a bind variable or to an `IN` or `IN/OUT` argument
- For reading, it fetches data from an `OUT` or `IN/OUT` argument

The `getStream()` method of the `ResultSet` class returns a stream object that can be used for reading data.

The `status()` method of these classes determines the status of the streaming operation.

### Binding Data in Streaming Mode; SELECT/DML and PL/SQL

To bind data in a streaming mode, follow these steps and review Example 3–24:

1. Create a `SELECT/DML` or PL/SQL statement with appropriate bind placeholders.
2. Call the `setBinaryStreamMode()` or `setCharacterStreamMode()` method of the `Statement` class for each bind position that is used in the streaming mode. If the bind position is a PL/SQL `IN` or `IN/OUT` argument type, indicate this by calling the three-argument versions of these methods and setting the `inArg` parameter to `TRUE`.
3. Execute the statement; the `status()` method of the `Statement` class returns `NEEDS_STREAM_DATA`.
4. Obtain the stream object through a `getStream()` method of the `Statement` class.
5. Use `writeBuffer()` and `writeLastBuffer()` methods of the `Stream` class to write data.
6. Close the stream with `closeStream()` method of the `Statement` class.
7. After all streams are closed, the `status()` method of the `Statement` class changes to an appropriate value, such as `UPDATE_COUNT_AVAILABLE`.

#### Example 3–24 How to Bind Data in a Streaming Mode

```c++
Statement *stmt = conn->createStatement("Insert Into testtab(longcol) values (:1)"); //longcol is LONG type column
stmt->setCharacterStreamMode(1, 100000);
stmt->executeUpdate();
Stream *instream = stmt->getStream(1);
char buffer[1000];
instream->writeBuffer(buffer, len); //write data
instream->writeLastBuffer(buffer, len); //repeat
stmt->closeStream(instream); //stmt->status() is
```

#### Fetching Data in Streaming Mode: PL/SQL

To fetch data from a streaming mode, follow these steps and review Example 3–25:

1. Create a `SELECT/DML` statement with appropriate bind placeholders.
2. Call the `setBinaryStreamMode()` or `setCharacterStreamMode()` method of the `Statement Class` for each bind position into which data is retrieved from the streaming mode.

3. Execute the statement; the `status()` method of the `Statement Class` returns `STREAM_DATA_AVAILABLE`.

4. Obtain the stream object through a `getStream()` method of the `Statement Class`.

5. Use `readBuffer()` and `readLastBuffer()` methods of the `Stream Class` to read data.

6. Close the stream with `closeStream()` method of the `Statement Class`.

**Example 3–25  How to Fetch Data in a Streaming Mode Using PL/SQL**

```
Statement *stmt = conn->createStatement("BEGIN testproc(:1); END; ");
    //argument 1 is OUT type
stmt->setCharacterStreamMode(1, 100000);
stmt->execute();
Stream *outarg = stmt->getStream(1);
    //use Stream::readBuffer/readLastBuffer to read data
```

**Fetching Data in Streaming Mode: ResultSet**

`Executing SQL Queries` and `Example 3–28` on page 3-20 provide an explanation of how to use the streaming interface with result sets.

**Working with Multiple Streams**

If you must work with multiple read and write streams, you must ensure that the read or write of one stream is completed before reading or writing on another stream. To determine stream position, use the `getCurrentStreamParam()` method of the `Statement Class` or the `getCurrentStreamColumn()` method of the `ResultSet Class`. The `status()` method of the `Stream Class` returns `READY_FOR_READ` if there is data in the stream available for reading, or it returns `INACTIVE` if all the data has been read, as described in Table 13–45. The application can then read the next streaming column.

`Example 3–26` demonstrates how to read and write with two concurrent streams. Note that it is not possible to use these streaming interfaces with the `setDataBuffer()` method in the same `Statement` and `ResultSet` objects.

**Example 3–26  How to Read and Write with Multiple Streams**

```
Statement *stmt = conn->createStatement{"Insert into testtab(longcol1, longcol2) values (:1,:2)"};
    //longcol1 AND longcol2 are 2 columns inserted in streaming mode
stmt->setBinaryStreamMode(1, 100000);
stmt->setBinaryStreamMode(2, 100000);
stmt->executeUpdate();
Stream *col1 = stmt->getStream(1);
Stream *col2 = stmt->getStream(2);
   col1->writeBuffer(buffer, len);        //first stream
   ...                                    //complete writing coll stream
   col1->writeLastBuffer(buffer, len);    //finish first stream and move to col2
   col2->writeBuffer(buffer, len);        //second stream
```
//reading multiple streams
stmt = conn->createStatement("select longcol1, longcol2 from testtab");
ResultSet *rs = stmt->executeQuery();
rs->setBinaryStreamMode(1, 100000);
rs->setBinaryStreamMode(2, 100000);

while (rs->next())
{
    Stream *s1 = rs->getStream(1)
    while (s1->status() == Stream::READY_FOR_READ)
    {
        s1->readBuffer(buffer, size);  //process
    }  //first streaming column done
    rs->closeStream(s1);

    //move onto next column. rs->getCurrentStreamColumn() returns 2
    Stream *s2 = rs->getStream(2)
    while (s2->status() == Stream::READY_FOR_READ)
    {
        s2->readBuffer(buffer, size);  //process
    }  //close the stream
    rs->closeStream(s2);
}

See Also: "Application-Managed Data Buffering" on page 12-8

Modifying Rows Iteratively

While you can issue the executeUpdate method repeatedly for each row, OCCI provides an efficient mechanism for sending data for multiple rows in a single network round-trip. Use the addIteration() method of the Statement class to perform batch operations that modify a different row with each iteration.

To execute INSERT, UPDATE, and DELETE operations iteratively, you must:
- Set the maximum number of iterations
- Set the maximum parameter size for variable length parameters

Setting the Maximum Number of Iterations

For iterative execution, first specify the maximum number of iterations that would be done for the statement by calling the setMaxIterations() method:

Statement->setMaxIterations(int maxIterations);

You can retrieve the current maximum iterations setting by calling the getMaxIterations() method.

Setting the Maximum Parameter Size

If the iterative execution involves variable-length data types, such as string and Bytes, then you must set the maximum parameter size so that OCCI can allocate the maximum size buffer:

Statement->setMaxParamSize(int parameterIndex, int maxParamSize);

You do not have to set the maximum parameter size for fixed-length data types, such as Number and Date, or for parameters that use the setDataBuffer() method.
You can retrieve the current maximum parameter size setting by calling the getMaxParamSize() method.

**Executing an Iterative Operation**

Once you have set the maximum number of iterations and (if necessary) the maximum parameter size, iterative execution using a parameterized statement is straightforward, as shown in Example 3–27.

Iterative execution is designed only for use in INSERT, UPDATE and DELETE operations that use either standard or parameterized statements. It cannot be used for callable statements and queries. Note that the data type cannot be changed between iterations. For example, if you use setInt() for parameter 1, then you cannot use setString() for the same parameter in a later iteration.

**Example 3–27  How to Execute an Iterative Operation**

```cpp
stmt->setSQL("INSERT INTO basket_tab VALUES(:1, :2);");

stmt->setString(1, "Apples");  // value for first parameter of first row
stmt->setInt(2, 6);             // value for second parameter of first row
stmt->addIteration();          // add the iteration

stmt->setString(1, "Oranges"); // value for first parameter of second row
stmt->setInt(1, 4);             // value for second parameter of second row
stmt->executeUpdate();          // execute statement
```

As shown in the example, you call the addIteration() method after each iteration except the last, after which you invoke executeUpdate() method. Of course, if you did not have a second row to insert, then you would not have to call the addIteration() method or make the subsequent calls to the setxxx() methods.

**Executing SQL Queries**

SQL query statements allow your applications to request information from a database based on any constraints specified. A result set is returned by the query.

**Using the Result Set**

Execution of a database query puts the results of the query into a set of rows called the result set. In OCCI, a SQL SELECT statement is executed by the executeQuery method of the Statement class. This method returns a ResultSet object that represents the results of a query.

```cpp
ResultSet *rs = stmt->executeQuery("SELECT * FROM basket_tab");
```

Once you have the data in the result set, you can perform operations on it. For example, suppose you wanted to print the contents of this table. The next() method of the ResultSet is used to fetch data, and the getxxx() methods are used to retrieve the individual columns of the result set, as shown in the following code example:

```cpp
cout << "The basket has:
while (rs->next())
{
    string fruit = rs->getString(1);  // get the first column as string
    int quantity = rs->getInt(2);     // get the second column as int
```
cout << quantity << " " << fruit << endl;
}

The `next()` and `status()` methods of the `ResultSet` class return `Status`, as defined in Table 13–38.

If data is available for the current row, then the status is `DATAAVAILABLE`. After all the data has been read, the status changes to `END_OF_FETCH`. If there are any output streams to be read, then the status is `STREAMDATAAVAILABLE`, until all the streamed data are read successfully.

Example 3–28 illustrates how to fetch streaming data into a result set, while section "Streamed Reads and Writes" on page 3-15 provides the general background.

Example 3–28 How to Fetch Data in Streaming Mode Using ResultSet

```cpp
char buffer[4096];
ResultSet *rs = stmt->executeQuery
    ("SELECT col1, col2 FROM tab1 WHERE col1 = 11");
rs->setCharacterStreamMode(2, 10000);

while (rs->next ()){
    unsigned int length = 0;
    unsigned int size = 500;
    Stream *stream = rs->getStream (2);
    while (stream->status () == Stream::READY_FOR_READ)
    {
        length += stream->readBuffer (buffer + length, size);
    }
    cout << "Read " << length << " bytes into the buffer" << endl;
}
```

Specifying the Query

The `IN` bind variables can be used with queries to specify constraints in the `WHERE` clause of a query. For example, the following program prints only those items that have a minimum quantity of 4:

```cpp
stmt->setSQL("SELECT * FROM basket_tab WHERE quantity >= :1");
int minimumQuantity = 4;
stmt->setInt(1, minimumQuantity); // set first parameter
ResultSet *rs = stmt->executeQuery();
cout << "The basket has:" << endl;

while (rs->next ()){
    cout << rs->getInt(2) << " " << rs->getString(1) << endl;
}
```

Optimizing Performance by Setting Prefetch Count

Although the `ResultSet` method retrieves data one row at a time, the actual fetch of data from the server need not entail a network round-trip for each row queried. To maximize the performance, you can set the number of rows to prefetch in each round-trip to the server.

You effect this either by setting the number of rows to be prefetched through the `setPrefetchRowCount()` method, or by setting the memory size to be used for prefetching through the `setPrefetchMemorySize()` method.
If you set both of these attributes, then the specified number of rows are prefetched unless the specified memory limit is reached first. If the specified memory limit is reached first, then the prefetch returns as many rows as can fit in the memory space defined by the call to the `setPrefetchMemorySize()` method.

By default, prefetching is turned on and the database fetches an extra row all the time. To turn prefetching off, set both the prefetch row count and memory size to 0.

Prefetching is not in effect if `LONG`, `LOB` or Opaque Type columns (such as `XMLType`) are part of the query.

### Executing Statements Dynamically

When you know that you must execute a DML operation, you use the `executeUpdate` method. Similarly, when you know that you must execute a query, you use `executeQuery()` method.

If your application must allow for dynamic events and you cannot be sure of which statement must be executed at run time, then OCCI provides the `execute()` method. Invoking the `execute()` method returns one of the following statuses:

- **UNPREPARED**
- **PREPARED**
- **RESULT_SET_AVAILABLE**
- **UPDATE_COUNT_AVAILABLE**
- **NEEDS_STREAM_DATA**
- **STREAM_DATA_AVAILABLE**

While invoking the `execute()` method returns one of these statuses, you can further 'interrogate' the statement by using the `status()` method.

```cpp
Statement stmt = conn->createStatement();
Statement::Status status = stmt->status();       // status is UNPREPARED
stmt->setSQL("select * from emp");
status = stmt->status();                         // status is PREPARED
```

If a statement object is created with a SQL string, then it is created in a **PREPARED** state. For example:

```cpp
Statement stmt = conn->createStatement("insert into foo(id) values(99)");
Statement::Status status = stmt->status();       // status is PREPARED
status = stmt->execute();                       // status is UPDATE_COUNT_AVAILABLE
```

When you set another SQL statement on the `Statement`, the status changes to **PREPARED**. For example:

```cpp
stmt->setSQL("select * from emp");              // status is PREPARED
status = stmt->execute();                      // status is RESULT_SET_AVAILABLE
```

### Statement Status Definitions

This section describes the possible values of `Status` related to a statement object:

- **UNPREPARED**
- **PREPARED**
- **RESULT_SET_AVAILABLE**
Executing Statements Dynamically

- **UPDATE_COUNT_AVAILABLE**
- **NEEDS_STREAM_DATA**
- **STREAM_DATA_AVAILABLE**

**UNPREPARED**
If you have not used the `setSQL()` method to attribute a SQL string to a statement object, then the statement is in an **UNPREPARED** state.

```cpp
Statement stmt = conn->createStatement();
Statement::Status status = stmt->status(); // status is UNPREPARED
```

**PREPARED**
If a Statement is created with an SQL string, then it is created in a **PREPARED** state. For example:

```cpp
Statement stmt = conn->createStatement("INSERT INTO demo_tab(id) VALUES(99)";
Statement::Status status = stmt->status();  // status is PREPARED
```

Setting another SQL statement on the `Statement` changes the status to **PREPARED**. For example:

```cpp
status = stmt->execute();                // status is UPDATE_COUNT_AVAILABLE
stmt->setSQL("SELECT * FROM demo_tab"); // status is PREPARED
```

**RESULT_SET_AVAILABLE**
A status of **RESULT_SET_AVAILABLE** indicates that a properly formulated query has been executed and the results are accessible through a result set.

When you set a statement object to a query, it is **PREPARED**. Once you have executed the query, the statement changes to **RESULT_SET_AVAILABLE**. For example:

```cpp
stmt->setSQL("SELECT * from EMP");      // status is PREPARED
status = stmt->execute();               // status is RESULT_SET_AVAILABLE
```

To access the data in the result set, issue the following statement:

```cpp
ResultSet *rs = Statement->getResultSet();
```

**UPDATE_COUNT_AVAILABLE**
When a DDL or DML statement in a **PREPARED** state is executed, its state changes to **UPDATE_COUNT_AVAILABLE**, as shown in the following code example:

```cpp
Statement stmt = conn->createStatement("INSERT INTO demo_tab(id) VALUES(99)";
Statement::Status status = stmt->status(); // status is PREPARED
status = stmt->execute();                  // status is UPDATE_COUNT_AVAILABLE
```

This status refers to the number of rows affected by the execution of the statement. It indicates that:

- The statement did not include any input or output streams.
- The statement was not a query but either a DDL or DML statement.

You can obtain the number of rows affected by issuing the following statement:

```cpp
Statement->getUpdateCount();
```

Note that a DDL statement results in an update count of zero (0). Similarly, an update that does not meet any matching conditions also produces a count of zero (0). In such

---

3-22  Oracle C++ Call Interface Programmer's Guide
a case, you cannot determine the kind of statement that has been executed from the reported status.

**NEEDS_STREAM_DATA**
If there are any output streams to be written, the execute does not complete until all the stream data is completely provided. In this case, the status changes to NEEDS_STREAM_DATA to indicate that a stream must be written. After writing the stream, call the status() method to find out if more stream data should be written, or whether the execution has completed.

In cases where your statement includes multiple streamed parameters, use the getCurrentStreamParam() method to discover which parameter must be written.

If you are performing an iterative or array execute, the getCurrentStreamIteration() method reveals to which iteration the data is to be written.

Once all the stream data has been processed, the status changes to either RESULT_SET_AVAILABLE or UPDATE_COUNTAVAILABLE.

**STREAM_DATA_AVAILABLE**
This status indicates that the application requires some stream data to be read in OUT or IN/OUT parameters before the execution can finish. After reading the stream, call the status method to find out if more stream data should be read, or whether the execution has completed.

In cases in which your statement includes multiple streamed parameters, use the getCurrentStreamParam() method to discover which parameter must be read.

If you are performing an iterative or array execute, then the getCurrentStreamIteration() method reveals from which iteration the data is to be read.

Once all the stream data has been handled, the status changes to UPDATE_COUNT_REMOVE_AVAILABLE.

The ResultSet class also has readable streams and it operates similar to the readable streams of the Statement class.

**Committing a Transaction**
All SQL DML statements are executed in the context of a transaction. An application causes the changes made by these statement to become permanent by either committing the transaction, or undoing them by performing a rollback. While the SQL COMMIT and ROLLBACK statements can be executed with the executeUpdate() method, you can also call the Connection::commit() and Connection::rollback() methods.

If you want the DML changes that were made to be committed immediately, you can turn on the auto commit mode of the Statement class by issuing the following statement:

```csharp
Statement::setAutoCommit(TRUE);
```

Once auto commit is in effect, each change is automatically made permanent. This is similar to issuing a commit right after each execution.

To return to the default mode, auto commit off, issue the following statement:

```csharp
Statement::setAutoCommit(FALSE);
```
Caching Statements

The statement caching feature establishes and manages a cache of statements within a session. It improves performance and scalability of application by efficiently using prepared cursors on the server side and eliminating repetitive statement parsing.

Statement caching can be used with connection and session pooling, and also without connection pooling. Please review Example 3–29 and Example 3–30 for typical usage scenarios.

Example 3–29  Statement Caching without Connection Pooling

These steps and accompanying pseudocode implement the statement caching feature without use of connection pools:

1. Create a Connection by making a createConnection() call on the Environment object.
   
   Connection *conn = env->createConnection( 
                   username, password, connecstr);

2. Enable statement caching on the Connection object by using a nonzero size parameter in the setStmtCacheSize() call.
   
   conn->setStmtCacheSize(10);

   Subsequent calls to getStmtCacheSize() would determine the size of the cache, while setStmtCacheSize() call changes the size of the statement cache, or disables statement caching if the size parameter is set to zero.

3. Create a Statement by making a createStatement() call on the Connection object; the Statement is returned if it is in the cache, or a new Statement with a NULL tag is created for the user.
   
   Statement *stmt = conn->createStatement(sql);

   To retrieve a previously cached tagged statement, use the alternate form of the createStatement() method:

   Statement *stmt = conn->createStatement(sql, tag);

4. Use the statement to execute SQL commands and obtain results.

5. Return the statement to cache.
   
   conn->terminateStatement(stmt, tag);

   If you do not want to cache this statement, use the disableCaching() call and an alternate from of terminateStatement():

   stmt->disableCaching();
   conn->terminateStatement(stmt);

   If you must verify whether a statement has been cached, issue an isCached() call on the Connection object.

   You can choose to tag a statement at release time and then reuse it for another statement with the same tag. The tag is used to search the cache. An untagged statement, where tag is NULL, is a special case of a tagged statement. Two statements are considered different if they only differ in their tags, and if only one of them is tagged.

6. Terminate the connection.
Example 3–30  Statement Caching with Connection Pooling

These steps and accompanying pseudocode implement the statement caching feature with connection pooling.

Statement caching is enabled only for connection created after the setStmtCacheSize() call.

If statement caching is not enabled at the pool level, it can still be implemented for individual connections in the pool.

1. Create a ConnectionPool by making a call to the createConnectionPool() of the Environment object.

   ConnectionPool *conPool = env->createConnectionPool(
       username, password, connectstr,
       minConn, maxConn, incrConn);

   If using a StatelessConnectionPool, call createStatelessConnectionPool() instead. Subsequent operations are the same for ConnectionPool and StatelessConnectionPool objects.

   StatelessConnectionPool *conPool = env->createStatelessConnectionPool(
       username, password, connectstr,
       minConn, maxConn, incrConn, mode);

2. Enable statement caching for all Connections in the ConnectionPool by using a nonzero size parameter in the setStmtCacheSize() call.

   conPool->setStmtCacheSize(10);

   Subsequent calls to getStmtCacheSize() would determine the size of the cache, while setStmtCacheSize() call changes the size of the statement cache, or disables statement caching if the size parameter is set to zero.

3. Get a Connection from the pool by making a createConnection() call on the ConnectionPool object; the Statement is returned if it is in the cache, or a new Statement with a NULL tag is created for the user.

   Connection *conn = conPool->createConnection(username, password, connectstr);

   To retrieve a previously cached tagged statement, use the alternate form of the createStatement() method:

   Statement *stmt = conn->createStatement(sql, tag);

4. Create a Statement by making a createStatement() call on the Connection object; the Statement is returned if it is in the cache, or a new Statement with a NULL tag is created for the user.

   Statement *stmt = conn->createStatement(sql);

   To retrieve a previously cached tagged statement, use the alternate form of the createStatement() method:

   Statement *stmt = conn->createStatement(sql, tag);

5. Use the statement to execute SQL commands and obtain results.

6. Return the statement to cache.

   conn->terminateStatement(stmt, tag);
If you do not want to cache this statement, use the `disableCaching()` call and an alternate from of `terminateStatement()`:

```c++
stmt->disableCaching();
conn->terminateStatement(stmt);
```

If you must verify whether a statement has been cached, issue an `isCached()` call on the `Connection` object.

7. Release the connection `terminateConnection()`.

```c++
conPool->terminateConnection(conn);
```

## Handling Exceptions

Each OCCI method can generate an exception if it is not successful. This exception is of type `SQLException`. OCCI uses the C++ Standard Template Library (STL), so any exception that can be thrown by the STL can also be thrown by OCCI methods.

The STL exceptions are derived from the standard exception class. The `exception::what()` method returns a pointer to the error text. The error text is guaranteed to be valid during the catch block.

The `SQLException` class contains Oracle specific error numbers and messages. It is derived from the standard exception class, so it too can obtain the error text by using the `exception::what()` method.

In addition, the `SQLException` class has two methods it can use to obtain error information. The `getErrorCode()` method returns the Oracle error number. The same error text returned by `exception::what()` can be obtained by the `getMessage()` method. The `getMessage()` method returns an STL string so that it can be copied like any other STL string.

Based on your error handling strategy, you may choose to handle OCCI exceptions differently from standard exceptions, or you may choose not to distinguish between the two.

If you decide that it is not important to distinguish between OCCI exceptions and standard exceptions, your catch block might look similar to the following:

```c++
catch (exception &excp) {
    cerr << excp.what() << endl;
}
```

Should you decide to handle OCCI exceptions differently than standard exceptions, your catch block might look like the following:

```c++
catch (SQLException &sqlExcp) {
    cerr << sqlExcp.getErrorCode << ' ' << sqlExcp.getMessage() << endl;
} catch (exception &excp) {
    cerr << excp.what() << endl;
}
```

In the preceding catch block, SQL exceptions are caught by the first block and non-SQL exceptions are caught by the second block. If the order of these two blocks were to be reversed, SQL exceptions would never be caught. Since `SQLException` is
derived from the standard exception, the standard exception catch block would handle the SQL exception as well.

See Also:

- Description of a special feature for handling errors that arise during batch updates, described in section "Modifying Rows Iteratively" on page 12-11 in Chapter 12, "Optimizing Performance of OCCI Applications"
- Oracle Database Error Messages for more information about Oracle error messages.

Handling Null and Truncated Data

In general, OCCI does not cause an exception when the data value retrieved by using the `getxxx()` methods of the `ResultSet` class or `Statement` class is NULL or truncated. However, this behavior can be changed by calling the `setErrorOnNull()` method or `setErrorOnTruncate()` method. If the `setErrorxxx()` methods are called with `causeException=TRUE`, then an `SQLException` is raised when a data value is NULL or truncated.

The default behavior is not to raise an `SQLException`. A column or parameter value can also be NULL, as determined by a call to `isNull()` for a `ResultSet` or `Statement` object returning TRUE:

```c
rs->isNull(columnIndex);
stmt->isNull(paramIndex);
```

If the column or parameter value is truncated, it also returns `TRUE` as determined by a `isTruncated()` call on a `ResultSet` or `Statement` object:

```c
rs->isTruncated(columnIndex);
stmt->isTruncated(paramIndex);
```

For data retrieved through the `setDataBuffer()` method and `setDataBufferArray()` method, exception handling behavior is controlled by the presence or absence of indicator variables and return code variables as shown in Table 3–1, Table 3–2, and Table 3–3.

<table>
<thead>
<tr>
<th>Table 3–1  Normal Data - Not Null and Not Truncated</th>
<th>Return Code</th>
<th>Indicator - not provided</th>
<th>Indicator - provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not provided</td>
<td>error = 0</td>
<td>error = 0</td>
<td>indicator = 0</td>
</tr>
<tr>
<td>Provided</td>
<td>error = 0</td>
<td>error = 0</td>
<td>indicator = 0</td>
</tr>
<tr>
<td></td>
<td>return code = 0</td>
<td>return code = 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3–2  Null Data</th>
<th>Return Code</th>
<th>Indicator - not provided</th>
<th>Indicator - provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not provided</td>
<td><code>SQLException</code></td>
<td>error = 0</td>
<td>indicator = -1</td>
</tr>
<tr>
<td></td>
<td>error = 1405</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provided</td>
<td><code>SQLException</code></td>
<td>error = 0</td>
<td>indicator = -1</td>
</tr>
<tr>
<td></td>
<td>error = 1405</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>return code = 1405</td>
<td>return code = 1405</td>
<td></td>
</tr>
</tbody>
</table>
In Table 3–3, `data_len` is the actual length of the data that has been truncated if this length is less than or equal to `SB2MAXVAL`. Otherwise, the indicator is set to –2.
This chapter provides information on how to implement object-relational programming using the Oracle C++ Call Interface (OCCI).

This chapter contains these topics:

- Overview of Object Programming
- Working with Objects in OCCI
- Representing Objects in C++ Applications
- Developing an OCCI Object Application
- Migrating C++ Applications Using OCCI
- Overview of Associative Access
- Overview of Navigational Access
- Overview of Complex Object Retrieval
- Working with Collections
- Using Object References
- Deleting Objects from the Database
- Type Inheritance
- A Sample OCCI Application

Overview of Object Programming

OCCI supports both the associative and navigational style of data access. Traditionally, third-generation language (3GL) programs manipulate data stored in a database by using the **associative access** based on the associations organized by relational database tables. In associative access, data is manipulated by executing SQL statements and PL/SQL procedures. OCCI supports associative access to objects by enabling your applications to execute SQL statements and PL/SQL procedures on the database server without incurring the cost of transporting data to the client.

Object-oriented programs that use OCCI can also make use of **navigational access** that is a key aspect of this programming paradigm. Object relationships between objects are implemented as references (REPs). Typically, an object application that uses navigational access first retrieves one or more objects from the database server by issuing a SQL statement that returns REPs to those objects. The application then uses those REPs to traverse related objects, and perform computations on these other objects as required. Navigational access does not involve executing SQL statements, except to fetch the references of an initial set of objects. By using the OCCI APIs for
navigational access, your application can perform the following functions on Oracle objects:

- Creating, accessing, locking, deleting, copying and flushing objects
- Getting references to objects and navigating through the references

This chapter gives examples that show you how to create a persistent object, access an object, modify an object, and flush the changes to the database server. It discusses how to access the object using both navigational and associative approaches.

**Working with Objects in OCCI**

Many of the programming principles that govern a relational OCCI applications are identical for object-relational applications. An object-relational application uses the standard OCCI calls to establish database connections and process SQL statements. The difference is that the SQL statements that are issued retrieve object references, which can then be manipulated with OCCI object functions. An object can also be directly manipulated as a value (without using its object reference).

Instances of an Oracle type are categorized into **persistent objects** and **transient objects** based on their lifetime. Instances of persistent objects can be further divided into **standalone objects** and **embedded objects** depending on whether they are referenced by way of an object identifier.

This section discusses the following topics:

- Persistent Objects
- Transient Objects
- Values

**Persistent Objects**

A **persistent object** is an object which is stored in an Oracle database. It may be fetched into the object cache and modified by an OCCI application. The lifetime of a persistent object can exceed that of the application which is accessing it. There are two types of persistent objects:

- A **standalone instance** is stored in a database table row, and has a unique object identifier. An OCCI application can retrieve a reference to a standalone object, pin the object, and navigate from the pinned object to other related objects. Standalone objects may also be referred to as **referenceable objects**.

  It is also possible to select a persistent object, in which case you fetch the object by value instead of fetching it by reference.

- An **embedded instance** is not stored in a database table row, but rather is embedded within another object. Examples of embedded objects are objects which are attributes of another object, or objects that exist in an object column of a database table. Embedded objects do not have object identifiers, and OCCI applications cannot get REFs to embedded instances.

  Embedded objects may also be referred to as **nonreferenceable objects** or **value instances**. You may sometimes see them referred to as **values**, which is not to be confused with scalar data values. The context should make the meaning clear.

Users do not have to explicitly delete persistent objects that have been materialized through references.
Users should delete persistent objects created by application when the transactions are rolled back.

The SQL examples, Example 4–1 and Example 4–2, demonstrate the difference between these two types of persistent objects.

**Example 4–1  Creating Standalone Objects**

Objects that are stored in the object table `person_tab` are standalone objects. They have object identifiers and can be referenced. They can be pinned in an OCCI application.

```
CREATE TYPE person_t AS OBJECT
    (name      varchar2(30),
     age       number(3));
CREATE TABLE person_tab OF person_t;
```

**Example 4–2  Creating Embedded Objects**

Objects which are stored in the `manager` column of the `department` table are embedded objects. They do not have object identifiers, and they cannot be referenced. Therefore, they cannot be pinned in an OCCI application, and they also never have to be unpinned. They are always retrieved into the object cache by value.

```
CREATE TABLE department
    (deptno     number,
     deptname   varchar2(30),
     manager    person_t);
```

The Array Pin feature allows a vector of references to be dereferenced in one round-trip to return a vector of the corresponding objects. A new global method, `pinVectorOfRefs()`, takes a vector of `Refs` and populates a vector of `PObjects` in a single round-trip, saving the cost of pinning \( n-1 \) references in \( n-1 \) round-trips.

### Transient Objects

A transient object is an instance of an object type. Its lifetime cannot exceed that of the application. The application can also delete a transient object at any time.

The Object Type Translator (OTT) utility generates two `operator new` methods for each C++ class, as demonstrated in Two Methods for Operator new() in the Object Type Translator Utility: Example 4–3:

**Example 4–3  Two Methods for Operator new() in the Object Type Translator Utility**

```
class Person : public PObject {
    ...
public:
    dvoid *operator new(size_t size);    // creates transient instance
    dvoid *operator new(size_t size, Connection &conn, string table);
        // creates persistent instance
}
```

**Example 4–4** demonstrates how to dynamically create a transient object. Transient objects cannot be converted to persistent objects. Their role is fixed at the time they are instantiated, and it is your responsibility to free memory by deleting transient objects.
Example 4–4  How to Dynamically Create a Transient Object

Person *p = new Person();

A transient object can also be created on the stack as a local variable, as demonstrated in Example 4–5. The latter approach guarantees that the transient object is destroyed when the scope of the variable ends.

Example 4–5  How to Create a Transient Object as a Local Variable

Person p;

See Also:

Oracle Database Concepts for more information about objects

Values

In this manual, a value refers to either:

■ A scalar value which is stored in a non-object column of a database table. An OCCI application can fetch values from a database by issuing SQL statements.

■ An embedded (nonreferenceable) object.

The context should make it clear which meaning is intended.

It is possible to SELECT a referenceable object into the object cache, rather than pinning it, in which case you fetch the object by value instead of fetching it by reference.

Representing Objects in C++ Applications

Before an OCCI application can work with object types, those types must exist in the database. Typically, you create types with SQL DDL statements, such as CREATE TYPE.

This section discusses the following topics:

■ Creating Persistent and Transient Objects

■ Creating Object Representations using the OTT Utility

Creating Persistent and Transient Objects

This section discusses how persistent and transient objects are created.

Before you create a persistent object, you must have created the environment and opened a connection.

A persistent object is created in the database only when one of the following occurs:

■ The transaction is committed (Connection::commit())

■ The object cache is flushed (Connection::flushCache())

■ The object itself is flushed (PObject::flush())

Example 4–6 shows how to create a persistent object, addr, in the database table, addr_tab.

Example 4–6  How to Create a Persistent Object

CREATE TYPE ADDRESS AS OBJECT {
    state CHAR(2),
}
Representing Objects in C++ Applications

```cpp
zip_code CHAR(5));
CREATE TABLE ADDR_TAB of ADDRESS;
ADDRESS *addr = new(conn, "ADDR_TAB") ADDRESS("CA", "94065");
```

Example 4–7 shows hot to create an instance of the transient object ADDRESS.

**Example 4–7  How to Create a Transient Object**

```cpp
ADDRESS *addr_trans = new ADDRESS("MD", "94111");
```

Creating Object Representations using the OTT Utility

When your C++ application retrieves instances of object types from the database, it must have a client-side representation of the objects. The Object Type Translator (OTT) utility generates C++ class representations of database object types for you. Example 4–8 shows the declaration of a custom type in the database, and the corresponding C++ class that the OTT utility generates.

**Example 4–8  How to Declare a Custom Type in the Database**

```sql
CREATE TYPE address AS OBJECT (state CHAR(2), zip_code CHAR(5));
```

The OTT utility produces the following C++ class:

```cpp
class ADDRESS : public PObject {
    protected:
        string state;
        string zip;

    public:
        void *operator new(size_t size);
        void *operator new(size_t size,
            const Connection* conn,
            const string& table);
        string getSQLTypeName() const;
        void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
            unsigned int &schemaNameLen, void **typeName,
            unsigned int &typeNameLen) const;
        ADDRESS(void *ctx) : PObject(ctx) { };
        static void *readSQL(void *ctx);
        virtual void readSQL(AnyData& stream);
        static void writeSQL(void *obj, void *ctx);
        virtual void writeSQL(AnyData& stream);
};
```

These class declarations in Example 4–8 are automatically written by OTT to a header file that you name. This header file is included in the source files for an application to provide access to objects. Instances of a PObject (and also instances of classes derived from PObject) can be either transient or persistent. The methods readSQL() and writeSQL() are used internally by the OCCI object cache to linearize and delinearize the objects and are not to be used or modified by OCCI clients.

**See Also:** Chapter 8, "Object Type Translator Utility" for more information about the OTT utility.
Developing an OCCI Object Application

This section discusses the steps involved in developing a basic OCCI object application.

This section discusses the following topics:

- Basic Object Program Structure
- Basic Object Operational Flow

Basic Object Program Structure

The basic structure of an OCCI application that uses objects is similar to a relational OCCI application, the difference being object functionality. The steps involved in an OCCI object program include:

1. Initialize the Environment. Initialize the OCCI programming environment in object mode. Your application must include C++ class representations of database objects in a header file. You can create these classes by using the Object Type Translator (OTT) utility, as described in Chapter 8, "Object Type Translator Utility".

2. Establish a Connection. Use the environment handle to establish a connection to the database server.

3. Prepare a SQL statement. This is a local (client-side) step, which may include binding placeholders. In an object-relational application, this SQL statement should return a reference (REF) to an object.

4. Access the object.
   a. Associate the prepared statement with a database server, and execute the statement.
   b. By using navigational access, retrieve an object reference (REF) from the database server and pin the object. You can then perform some or all of the following:
      - Manipulate the attributes of an object and mark it as dirty (modified)
      - Follow a reference to another object or series of objects
      - Access type and attribute information
      - Navigate a complex object retrieval graph
      - Flush modified objects to the database server
   c. By using associative access, you can fetch an entire object by value by using SQL. Alternately, you can select an embedded (nonreferenceable) object. You can then perform some or all of the following:
      - Insert values into a table
      - Modify existing values

5. Commit the transaction. This step implicitly writes all modified objects to the database server and commits the changes.

6. Free statements and handles; the prepared statements should not be used or executed again.
Basic Object Operational Flow

Figure 3-1 shows a simple program logic flow for how an application might work with objects. For simplicity, some required steps are omitted.

**Figure 4–1 Basic Object Operational Flow**

The steps shown in Figure 3-1 are discussed in the following sections:

**Initialize OCCI in Object Mode**
If your OCCI application accesses and manipulates objects, then it is essential that you specify a value of `OBJECT` for the `mode` parameter of the `createEnvironment()` method, the first call in any OCCI application. Specifying this value for `mode` indicates to OCCI that your application works with objects. This notification has the following important effects:

- The object run-time environment is established.
- The object cache is set up.

Note that the `mode` parameter is not set to `OBJECT`, any attempt to use an object-related function results in an error.

The following code example demonstrates how to specify the `OBJECT` mode when creating an OCCI environment:

```c
Environment *env;
```

See Also:
- Chapter 3, "Relational Programming" for information about using OCCI to connect to a database server, process SQL statements, and allocate handles
- Chapter 8, "Object Type Translator Utility" for information about the OTT utility
- Chapter 13, "OCCI Application Programming Interface" for descriptions of OCCI relational functions and the `Connect` class and the `getMetaData()` method
Connection *con;
Statement *stmt;

env = Environment::createEnvironment(Environment::OBJECT);
con = Connection(userName, password, connectString);

Your application does not have to allocate memory when database objects are loaded into the object cache. The object cache provides transparent and efficient memory management for database objects. When database objects are loaded into the object cache, they are transparently mapped into the host language (C++) representation.

The object cache maintains the association between the object copy in the object cache and the corresponding database object. Upon commit, changes made to the object copy in the object cache are automatically propagated back to the database.

The object cache maintains a look-up table for mapping references to objects. When an application dereferences a reference to an object and the corresponding object is not yet cached in the object cache, the object cache automatically sends a request to the database server to fetch the object from the database and load it into the object cache. Subsequent dereferences of the same reference are faster since they are to the object cache itself and do not incur a round-trip to the database server.

Subsequent dereferences of the same reference fetch from the cache instead of requiring a round-trip. The exception to this is in a dereferencing operation that occurs just after a commit. In this case, the latest object copy from the server is returned. This ensures that the latest object from the database is cached after each transaction.

The object cache maintains a pin count for each persistent object in the object cache. When an application dereferences a reference to an object, the pin count of the object is incremented. The subsequent dereferencing of the same reference to the object does not change the pin count. Until the reference to the object goes out of scope, the object continues to be pinned in the object cache and be accessible by the OCCI client.

The pin count functions as a reference count for the object. The pin count of the object becomes zero (0) only when there are no more references referring to this object, during which time the object becomes eligible for garbage collection. The object cache uses a least recently used algorithm to manage the size of the object cache. This algorithm frees objects with a pin count of 0 when the object cache reaches the maximum size.

**Pin Object**

In most situations, OCCI users do not have to explicitly pin or unpin the objects because the object cache automatically keeps track of the pin counts of all the objects in the cache. As explained earlier, the object cache increments the pin count when a reference points to the object and decrements it when the reference goes out of scope or no longer points to the object.

But there is one exception. If an OCCI application uses `Ref<T>::ptr()` method to get a pointer to the object, then the `pin` and `unpin` methods of the `PObject` class can be used by the application to control pinning and unpinning of the objects in the object cache.

**Operate on Object in Cache**

Note that the object cache does not manage the contents of object copies; it does not automatically refresh object copies. Your application must ensure the validity and consistency of object copies.
Flush Changes to Object
Whenever changes are made to object copies in the object cache, your application is responsible for flushing the changed object to the database.

Memory for the object cache is allocated on demand when objects are loaded into the object cache.

The client-side object cache is allocated in the program's process space. This object cache is the memory for objects that have been retrieved from the database server and are available to your application.

If you initialize the OCCI environment in object mode, your application allocates memory for the object cache, whether the application actually uses object calls.

There is only one object cache allocated for each OCCI environment. All objects retrieved or created through different connections within the environment use the same physical object cache. Each connection has its own logical object cache.

Deletion of an Object
For objects retrieved into the cache by dereferencing a reference, you should not perform an explicit delete. For such objects, the pin count is incremented when a reference is dereferenced for the first time and decremented when the reference goes out of scope. When the pin count of the object becomes 0, indicating that all references to that object are out of scope, the object is automatically eligible for garbage collection and subsequently deleted from the cache.

For persistent objects that have been created by calling the `new` operator, you must call a `delete` if you do not commit the transaction. Otherwise, the object is garbage collected after the commit. This is because when such an object is created using `new`, its pin count is initially 0. However, because the object is dirty it remains in the cache. After a commit, it is no longer dirty and thus garbage collected. Therefore, a delete is not required.

If a commit is not performed, then you must explicitly call `delete` to destroy that object. You can do this if there are no references to that object. For transient objects, you must delete explicitly to destroy the object.

You should not call a delete operator on a persistent object. A persistent object that is not marked/dirty is freed by the garbage collector when its pin count is 0. However, for transient objects you must delete explicitly to destroy the object.

Migrating C++ Applications Using OCCI
This section describes how to migrate existing C++ applications using OCCI.

The steps of migration are:
1. Determine object model and class hierarchy
2. Use JDeveloper9i to map to Oracle object schema
3. Generate C++ header files using Oracle Type Translator
4. Modify old C++ access classes as required to work with new object type definitions
5. Add functionality for transient and persistent object management, as required.
Overview of Associative Access

You can employ SQL within OCCI to retrieve objects, and to perform DML operations. This section discusses the following topics:

- Using SQL to Access Objects
- Inserting and Modifying Values

**See Also:** complete code listing of the demonstration programs

### Using SQL to Access Objects

In the previous sections we discussed navigational access, where SQL is used only to fetch the references of an initial set of objects and then navigate from them to the other objects. Here we discuss how to fetch the objects using SQL.

The following example shows how to use the `ResultSet::getObject()` method to fetch objects through associative access where it gets each object from the table, `addr_tab`, using SQL:

```cpp
string sel_addr_val = "SELECT VALUE(address) FROM ADDR_TAB address";
ResultSet *rs = stmt->executeQuery(sel_addr_val);
while (rs->next())
{
    ADDRESS *addr_val = rs->getObject(1);
    cout << "state: " << addr_val->getState();
}
```

The objects fetched through associative access are termed value instances and they behave just like transient objects. Methods such as `markModified()`, `flush()`, and `markDeleted()` are applicable only for persistent objects.

Any changes made to these objects are not reflected in the database. Since the object returned is a value instance, it is the user's responsibility to free memory by deleting the object pointer.

### Inserting and Modifying Values

We have just seen how to use SQL to access objects. OCCI also provides the ability to use SQL to insert new objects or modify existing objects in the database server through the `Statement::setObject` method interface.

The following example creates a transient object `Address` and inserts it into the database table `addr_tab`:

```cpp
ADDRESS *addr_val = new address("NV", "12563"); // new a transient instance
stmt->setSQL("INSERT INTO ADDR_TAB values(:1) ");
stmt->setObject(1, addr_val);
stmt->execute();
```

### Overview of Navigational Access

By using navigational access, you engage in a series of operations. This section discusses the following topics:

- Retrieving an Object Reference (REF) from the Database Server
Retrieving an Object Reference (REF) from the Database Server

To work with objects, your application must first retrieve one or more objects from the database server. You accomplish this by issuing a SQL statement that returns references (REFs) to one or more objects.

It is also possible for a SQL statement to fetch value instances, rather than REFs, from a database.

The following SQL statement retrieves a REF to a single object address from the database table addr_tab:

```c++
string sel_addr = 'SELECT REF(address)
FROM addr_tab address
WHERE zip_code = '94065'';
```

The following code example illustrates how to execute the query and fetch the REF from the result set.

```c++
ResultSet *rs = stmt->executeQuery(sel_addr);
rs->next();
Ref<address> addr_ref = rs->getRef(1);
```

At this point, you could use the object reference to access and manipulate the object or objects from the database.

See Also: complete code listing of the demonstration programs

Pinning an Object

This section deals with a simple pin operation involving a single object at a time. For information about retrieving multiple objects through complex object retrieval, see the section Overview of Complex Object Retrieval on page 4-14.

Upon completion of the fetch step, your application has a REF to an object. The actual object is not currently available to work with. Before you can manipulate an object, it must be pinned. Pinning an object loads the object into the object cache, and enables you to access and modify the object's attributes and follow references from that object to other objects. Your application also controls when modified objects are written back to the database server.

OCCI requires only that you dereference the REF in the same way you would dereference any C++ pointer. Dereferencing the REF transparently materializes the object as a C++ class instance.
Continuing the Address class example from the previous section, assume that the user has added the following method:

```c++
string Address::getState()
{
    return state;
}
```

To dereference this REF and access the object's attributes and methods:

```c++
string state = addr_ref->getState();  // -> pins the object
```

The first time `Ref<T>` (addr_ref) is dereferenced, the object is pinned, which is to say that it is loaded into the object cache from the database server. From then on, the behavior of operator `->` on `Ref<T>` is just like that of any C++ pointer (T *). The object remains in the object cache until the REF (addr_ref) goes out of scope. It then becomes eligible for garbage collection.

Now that the object has been pinned, your application can modify that object.

### Manipulating Object Attributes

Manipulating object attributes is no different from that of accessing them as shown in the previous section. Let us assume the Address class has the following user defined method that sets the state attribute to the input value:

```c++
void Address::setState(string new_state)
{
    state = new_state;
}
```

The following example shows how to modify the state attribute of the object, addr:

```c++
addr_ref->setState('PA');
```

As explained earlier, the first invocation of the operator `->` on `Ref<T>` loads the object, if it is not in the object cache.

### Marking Objects and Flushing Changes

In the example in the previous section, an attribute of an object was changed. This change exists only in the client-side cache; you must implement specific programmatic steps to write the changes to the database.

#### Marking an Object as Modified (Dirty)

The first step is to indicate that the object has been modified. This is done by calling the `markModified()` method on the object (derived method of `PObject`). This method marks the object as dirty (modified).

Continuing the previous example, after object attributes are manipulated, the object referred to by `addr_ref` can be marked dirty as follows:

```c++
addr_ref->markModified();
```

#### Recording Changes in the Database

Objects that have had their dirty flag set must be flushed to the database server for the changes to be recorded in the database. This can be done in three ways:
Flush a single object marked dirty by calling the method `flush`, a derived method of `PObject`.

Flush the entire object cache using the `Connection::flushCache()` method. In this case, OCeC traverses the dirty list maintained by the object cache and flushes all the dirty objects.

Commit a transaction by calling the `Connection::commit()` method. Doing so also traverses the dirty list and flushes the objects to the database server. The dirty list includes newly created persistent objects.

**Collecting Garbage in the Object Cache**

The object cache has two important associated parameters:

- The maximum cache size percentage
- The optimal cache size

These parameters refer to levels of cache memory usage, and they help determine when the cache automatically ‘ages out’ eligible objects to free up memory.

If the memory occupied by the objects currently in the cache reaches or exceeds the maximum cache size, the cache automatically begins to free (or age out) unmarked objects which have a pin count of zero. The cache continues freeing such objects until memory usage in the cache reaches the optimal size, or until it runs out of objects eligible for freeing. Note that the cache can grow beyond the specified maximum cache size.

The maximum object cache size (in bytes) is computed by incrementing the optimal cache size (optimal_size) by the maximum cache size percentage (max_size_percentage), as follows:

\[
\text{Maximum cache size} = \text{optimal}_\text{size} + \text{optimal}_\text{size} \times \frac{\text{max}_\text{size}_\text{percentage}}{100};
\]

The default value for the maximum cache size percentage is 10%. The default value for the optimal cache size is 8MB. When a persistent object is created through the overloaded `PObject::new()` operator, the newly created object is marked dirty and its pin count is set to 0.

These parameters can be set or retrieved using the following member functions of the `Environment` class:

- `void setCacheMaxSize(unsigned int maxSize);`
- `unsigned int getCacheMaxSize() const;`
- `void setCacheOptSize(unsigned int OptSize);`
- `unsigned int getCacheOptSize() const;`

"Pin Object" on page 4-8 describes how pin count of an object functions as a reference count and how an unmarked object with a 0 pin count can become eligible for garbage collection. For a newly created persistent object, the object is unmarked after the transaction is committed or aborted, and if the object has a 0 pin count. Because nothing is referencing this object, it becomes a candidate for ageing out.

If you are working with several object that have a large number of string or collection attributes, most of the memory is allocated from the C++ heap; this is because OCeC uses STLs. You should therefore set the cache size to a low value to avoid high memory use before garbage collection activates.
Ensuring Transactional Consistency of References

As described in the previous section, dereferencing a $\text{Ref}<T>$ for the first time results in the object being loaded into the object cache from the database server. From then on, the behavior of operator $->$ on $\text{Ref}<T>$ equals any C++ pointer, and it provides access to the object copy in the cache. But when the transaction commits or aborts, the object copy in the cache can no longer be valid because it could be modified by any other client. Therefore, after the transaction ends, when the $\text{Ref}<T>$ is again dereferenced, the object cache recognizes the fact that the object is no longer valid and fetches the most recent copy from the database server.

Overview of Complex Object Retrieval

In the examples discussed earlier, only a single object was fetched or pinned at a time. In these cases, each pin operation involved a separate database server round-trip to retrieve the object.

Object-oriented applications often model their problems as a set of interrelated objects that form graphs of objects. These applications process objects by starting with some initial set of objects and then using the references in these objects to traverse the remaining objects. In a client/server setting, each of these traversals could result in costly network round-trips to fetch objects.

The performance of such applications can be increased with complex object retrieval (COR). This is a prefetching mechanism in which an application specifies some criteria (content and boundary) for retrieving a set of linked objects in a single network round-trip. Using COR does not mean that these prefetched objects are pinned. They are fetched into the object cache, so that subsequent pin calls are local operations.

A complex object is a set of logically related objects consisting of a root object, and a set of objects each of which is prefetched based on a given depth level. The root object is explicitly fetched or pinned. The depth level is the shortest number of references that have to be traversed from the root object to a given prefetched object in a complex object.

An application specifies a complex object by describing its content and boundary. The fetching of complex objects is constrained by an environment’s prefetch limit, the amount of memory in the object cache that is available for prefetching objects.

The use of complex object retrieval does not add functionality; it only improves performance, and so its use is optional.

This section discusses the following topics:

- Retrieving Complex Objects
- Prefetching Complex Objects

Retrieving Complex Objects

An OCCI application can achieve COR by setting the appropriate attributes of a $\text{Ref}<T>$ before dereferencing it using the following methods:

```cpp
// prefetch attributes of the specified type name up to the specified depth
$\text{Ref}<T>$::setPrefetch(const string &typeName, unsigned int depth);
```
// prefetch all the attribute types up to the specified depth.
Ref<T>::setPrefetch(unsigned int depth);

The application can also choose to fetch all objects reachable from the root object by
way of REFS (transitive closure) to a certain depth. To do so, set the level parameter to
the depth desired. For the preceding two examples, the application could also specify
(PO object REF, OCCI_MAX_PREFETCH_DEPTH) and (PO object REF, 1)
respectively to prefetch required objects. Doing so results in many extraneous fetches
but is quite simple to specify, and requires only one database server round-trip.

As an example for this discussion, consider the following type declaration:

CREATE TYPE customer(...);
CREATE TYPE line_item(...);
CREATE TYPE line_item_varray as VARRAY(100) of REF line_item;
CREATE TYPE purchase_order AS OBJECT
  ( po_number         NUMBER,
    cust              REF customer,
    related_orders    REF purchase_order,
    line_items        line_item_varray);

The purchase_order type contains a scalar value for po_number, a VARRAY of
line_items, and two references. The first is to a customer type and the second is to
a purchase_order type, indicating that this type can be implemented as a linked
list.

When fetching a complex object, an application must specify the following:

■ A reference to the desired root object

■ One or more pairs of type and depth information to specify the boundaries of the
  complex object. The type information indicates which REF attributes should be
  followed for COR, and the depth level indicates how many levels deep those links
  should be followed.

In the case of the purchase_order object in the preceding example, the application
must specify the following:

■ The reference to the root purchase_order object

■ One or more pairs of type and depth information for customer, purchase_order, or line_item

An application prefetching a purchase order needs access to the customer information
for that purchase order. Using simple navigation, this would require two database
server accesses to retrieve the two objects.

Through complex object retrieval, customer can be prefetched when the application
pins the purchase_order object. In this case, the complex object would consist of the
purchase_order object and the customer object it references.

In the previous example, if the application wanted to prefetch a purchase order and
the related customer information, the application would specify the
purchase_order object and indicate that customer should be followed to a depth
level of one as follows:

Ref<PURCHASE_ORDER> poref;
poref.setPrefetch("CUSTOMER",1);

If the application wanted to prefetch a purchase order and all objects in the object
graph it contains, the application would specify the purchase_order object and
indicate that both customer and purchase_order should be followed to the maximum depth level possible as follows:

```cpp
Ref<PURCHASE_ORDER> poref;
poref.setPrefetch("CUSTOMER", OCCI_MAX_PREFETCH_DEPTH);
poref.setPrefetch("PURCHASE_ORDER", OCCI_MAX_PREFETCH_DEPTH);
```

where OCCI_MAX_PREFETCH_DEPTH specifies that all objects of the specified type reachable through references from the root object should be prefetched.

If an application wanted to prefetch a purchase order and all the line items associated with it, the application would specify the purchase_order object and indicate that line_items should be followed to the maximum depth level possible as follows:

```cpp
Ref<PURCHASE_ORDER> poref;
poref.setPrefetch("LINE_ITEM", 1);
```

### Prefetching Complex Objects

After specifying and fetching a complex object, subsequent fetches of objects contained in the complex object do not incur the cost of a network round-trip, because these objects have been prefetched and are in the object cache. Keep in mind that excessive prefetching of objects can lead to a flooding of the object cache. This flooding, in turn, may force out other objects that the application had pinned, leading to a performance degradation instead of performance improvement.

Note that if there is insufficient memory in the object cache to hold all prefetched objects, some objects may not be prefetched. The application then incurs a network round-trip when those objects are accessed later.

You must have the SELECT privilege for all prefetched objects. Objects in the complex object for which the application does not have SELECT privilege cannot be prefetched.

An entire vector ofRefs can be prefetched into object cache in a single round-trip by using the global pinVectorOfRefs() method of the Connection Class. This method reduces the number of round-trips for an n-sized vector ofRefs from n to 1, and tracks the newly pinned objects through an OUT parameter vector.

### Working with Collections

Oracle supports two kinds of collections - variable length arrays (ordered collections) and nested tables (unordered collections). OCCI maps both of them to a Standard Template Library (STL) vector container, giving you the full power, flexibility, and speed of an STL vector to access and manipulate the collection elements. Example 4–9 shows the SQL DDL to create a VARRAY and an object that contains an attribute of type VARRAY, and the resulting C++ declaration that OTT generates.

**Example 4–9  How to Create a VARRAY Collection**

```sql
CREATE TYPE ADDR_LIST AS VARRAY(3) OF REF ADDRESS;
CREATE TYPE PERSON AS OBJECT (name VARCHAR2(20), addr_l ADDR_LIST);
```

Here is the C++ class declaration generated by OTT:

```cpp
class PERSON : public PObject
{
protected:
  string name;
  vector< Ref< ADDRESS > > addr_l;
};
```
public:
    void *operator new(size_t size);
    void *operator new(size_t size,
        const Connection* conn,
        const string& table);
    string getSQLTypeName() const;
    void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
        unsigned int &schemaNameLen, void **typeName,
        unsigned int &typeNameLen) const;
    PERSON (void *ctx) : PObject(ctx) {);
    static void *readSQL(void *ctx);
    virtual void readSQL(AnyData& stream);
    static void writeSQL(void *obj, void *ctx);
    virtual void writeSQL(AnyData& stream);
};

See Also: complete code listing of the demonstration programs

Fetching Embedded Objects

If your application must fetch an embedded object, which is an object stored in a column of a regular table rather than an object table, you cannot use the REF retrieval mechanism. Embedded instances do not have object identifiers, so it is not possible to get a reference to them. Therefore, they cannot serve as the basis for object navigation. There are still many situations, however, in which an application fetches embedded instances.

For example, assume that an address type has been created.

```sql
CREATE TYPE address AS OBJECT
(   street1     varchar2(50),
   street2     varchar2(50),
   city        varchar2(30),
   state       char(2),
   zip         number(5));
```

You could then use that type as the data type of a column in another table:

```sql
CREATE TABLE clients
(   name       varchar2(40),
   addr        address);
```

Your OCCI application could then issue the following SQL statement:

```sql
SELECT addr FROM clients
WHERE name='BEAR BYTE DATA MANAGEMENT';
```

This statement would return an embedded address object from the clients table. The application could then use the values in the attributes of this object for other processing. The application should execute the statement and fetch the object in the same way as described in the section "Overview of Associative Access" on page 4-10.

Nullness

If a column in a row of a database table has no value, then that column is said to be NULL, or to contain a NULL. Two different types of NULLs can apply to objects:

- Any attribute of an object can have a NULL value. This indicates that the value of that attribute of the object is not known.
An object may be **atomically NULL**. Therefore, the value of the entire object is unknown.

Atomic **NULL**ness is different from nonexistence. An atomically **NULL** object still exists, its value is just not known. It may be thought of as an existing object with no data.

For every type of object attribute, OCCI provides a corresponding class. For instance, **NUMBER** attribute type maps to the **Number** class, **REF** maps to **RefAny**, and so on. Each and every OCCI class that represents a data type provides two methods:

- `isNull()` — returns whether the object is **NULL**
- `setNull()` — sets the object to **NULL**

Similarly, these methods are inherited from the **PObject** class by all the objects and can be used to access and set atomically **NULL** information about them.

### Using Object References

OCCI provides the application with the flexibility to access the contents of the objects using their pointers or their references. OCCI provides the **PObject**:`getRef()` method to return a reference to a persistent object. This call is valid for persistent objects only.

### Deleting Objects from the Database

OCCI users can use the overloaded **PObject**:`operator new()` to create the persistent objects. However, to delete the object from the database server, it is best to call **ref**:`markDelete()` directly on the **Ref**; this prevents the object from getting into the client cache. If the object is in the client cache, it can be removed by an **obj**:`markDelete()` call on the object. The object marked for deletion is permanently removed when the transaction commits.

### Type Inheritance

Type inheritance of objects has many similarities to inheritance in C++ and Java. You can create an object type as a subtype of an existing object type. The subtype is said to inherit all the attributes and methods (member functions and procedures) of the supertype, which is the original type. Only single inheritance is supported; an object cannot have multiple supertypes. The subtype can add new attributes and methods to the ones it inherits. It can also override (redefine the implementation) of any of its inherited methods. A subtype is said to extend (that is, inherit from) its supertype.

**See Also:** Oracle Database Object-Relational Developer’s Guide for a more complete discussion of this topic

As an example, a type **Person_t** can have a subtype **Student_t** and a subtype **Employee_t**. In turn, **Student_t** can have its own subtype, **PartTimeStudent_t**. A type declaration must have the flag **NOT FINAL** so that it can have subtypes. The default is **FINAL**, which means that the type can have no subtypes.

All types discussed so far in this chapter are **FINAL**. All types in applications developed before Oracle Database release 8.1.7 are **FINAL**. A type that is **FINAL** can be altered to be **NOT FINAL**. A **NOT FINAL** type with no subtypes can be altered to be **FINAL**. **Person_ t** is declared as **NOT FINAL** for our example:

```sql
CREATE TYPE Person_t AS OBJECT
(  ssn NUMBER,
```
A subtype inherits all the attributes and methods declared in its supertype. It can also declare new attributes and methods, which must have different names than those of the supertype. The keyword `UNDER` identifies the supertype, like this:

```
CREATE TYPE Student_t UNDER Person_t
    ( deptid NUMBER,
      major VARCHAR2(30)) NOT FINAL;
```

The newly declared attributes `deptid` and `major` belong to the subtype `Student_t`. The subtype `Employee_t` is declared as, for example:

```
CREATE TYPE Employee_t UNDER Person_t
    ( empid NUMBER,
      mgr VARCHAR2(30));
```

Subtype `Student_t` can have its own subtype, such as `PartTimeStudent_t`:

```
CREATE TYPE PartTimeStuden_t UNDER Student_t ( numhours NUMBER);
```

### Substitutability

The benefits of polymorphism derive partially from the property substitutability. Substitutability allows a value of some subtype to be used by code originally written for the supertype, without any specific knowledge of the subtype being needed in advance. The subtype value behaves to the surrounding code just like a value of the supertype would, even if it perhaps uses different mechanisms within its specializations of methods.

Instance substitutability refers to the ability to use an object value of a subtype in a context declared in terms of a supertype. `REF` substitutability refers to the ability to use a `REF` to a subtype in a context declared in terms of a `REF` to a supertype.

`REF` type attributes are substitutable, that is, an attribute defined as `REF T` can hold a `REF` to an instance of `T` or any of its subtypes.

Object type attributes are substitutable, that is, an attribute defined to be of (an object) type `T` can hold an instance of `T` or any of its subtypes.

Collection element types are substitutable, that is, if we define a collection of elements of type `T`, then it can hold instances of type `T` and any of its subtypes. Here is an example of object attribute substitutability:

```
CREATE TYPE Book_t AS OBJECT
    ( title VARCHAR2(30),
      author Person_t     /* substitutable */);
```

Thus, a `Book_t` instance can be created by specifying a title string and a `Person_t` (or any subtype of `Person_t`) object:

```
Book_t('My Oracle Experience',
    Employee_t(12345, 'Joe', 'SF', 1111, NULL))
```
NOT INSTANTIABLE Types and Methods

A type can be declared NOT INSTANTIABLE, which means that there is no constructor (default or user defined) for the type. Thus, it is not be possible to construct instances of this type. The typical usage would be to define instantiable subtypes for such a type. Here is how this property is used:

CREATE TYPE Address_t AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;
CREATE TYPE USAddress_t UNDER Address_t(...);
CREATE TYPE IntlAddress_t UNDER Address_t(...);

A method of a type can be declared to be NOT INSTANTIABLE. Declaring a method as NOT INSTANTIABLE means that the type is not providing an implementation for that method. Further, a type that contains any NOT INSTANTIABLE methods must necessarily be declared as NOT INSTANTIABLE. For example:

CREATE TYPE T AS OBJECT
(x NUMBER,
   NOT INSTANTIABLE MEMBER FUNCTION func1() RETURN NUMBER
) NOT INSTANTIABLE;

A subtype of NOT INSTANTIABLE can override any of the NOT INSTANTIABLE methods of the supertype and provide concrete implementations. If there are any NOT INSTANTIABLE methods remaining, the subtype must also necessarily be declared as NOT INSTANTIABLE.

A NOT INSTANTIABLE subtype can be defined under an instantiable supertype. Declaring a NOT INSTANTIABLE type to be FINAL is not useful and is not allowed.

OCCI Support for Type Inheritance

The following calls support type inheritance.

Connection::getMetaData()

This method provides information specific to inherited types. Additional attributes have been added for the properties of inherited types. For example, you can get the supertype of a type.

Bind and Define Functions

The setRef(), setObject() and setVector() methods of the Statement class are used to bind REF, object, and collections respectively. All these functions support REF, instance, and collection element substitutability. Similarly, the corresponding getxxx() methods to fetch the data also support substitutability.

OTT Support for Type Inheritance

Class declarations for objects with inheritance are similar to the simple object declarations except that the class is derived from the parent type class and only the fields corresponding to attributes not in the parent class are included. The structure for these declarations is listed in Example 4–10:

Example 4–10 OTT Support Inheritance

class <typename> : public <parentTypename>
{
  protected:
    <OCCItype1> <attributename1>;
    ...
}
A Sample OCCI Application

In this structure, all variables are the same as in the simple object case. parentTypename refers to the name of the parent type, that is, the class name of the type from which typename inherits.

Example 4–11 List of demo2.sql for a Sample OCCI Application

drop table ADDR_TAB
/
drop table PERSON_TAB
/
drop type STUDENT
/
drop type PERSON
/
drop type ADDRESS_TAB
/
drop type ADDRESS
/
drop type FULLNAME
/
CREATE TYPE FULLNAME AS OBJECT (first_name CHAR(20), last_name CHAR(20))
/
CREATE TYPE ADDRESS AS OBJECT (state CHAR(20), zip CHAR(20))
/
CREATE TYPE ADDRESS_TAB AS VARRAY(3) OF REF ADDRESS
/
CREATE TYPE PERSON AS OBJECT (id NUMBER, name FULLNAME, curr_addr REF ADDRESS, prev_addr_l ADDRESS_TAB) NOT FINAL
/
CREATE TYPE STUDENT UNDER PERSON (school_name CHAR(20))
/
CREATE TABLE ADDR_TAB OF ADDRESS
/
CREATE TABLE PERSON_TAB OF PERSON
/
Example 4–12  Listing of demo2.typ for a Sample OCCI Application

TYPE FULLNAME GENERATE CFullName as MyFullName
TYPE ADDRESS GENERATE CAddress as MyAddress
TYPE PERSON GENERATE CPerson as MyPerson
TYPE STUDENT GENERATE CStudent as MyStudent

Example 4–13  Listing of OTT Command that Generates Files for a Sample OCCI Application

OTT attempts to connect with username demousr; the system prompts for the password.

ottom useri=demousr intypedemo2.typ code=cpp hfile=demo2.h
cppfile=demo2.cpp mapfile=mappings.cpp attraccess=private

Example 4–14  Listing of mappings.h for a Sample OCCI Application

#ifndef MAPPINGS_ORACLE
#define MAPPINGS_ORACLE

#ifndef OCCI_ORACLE
#include <occi.h>
#endif

#ifndef DEMO2_ORACLE
#include "demo2.h"
#endif

void mappings(oracle::occi::Environment* envOCCI_);
#endif

Example 4–15  Listing of mappings.cpp for a Sample OCCI Application

#ifndef MAPPINGS_ORACLE
#include "mappings.h"
#endif

void mappings(oracle::occi::Environment* envOCCI_)
{
oracle::occi::Map *mapOCCI_ = envOCCI_->_getMap();
mapOCCI_->_put("HR.FULLNAME", &CFullName::readSQL, &CFullName::writeSQL);
mapOCCI_->_put("HR.ADDRESS", &CAddress::readSQL, &CAddress::writeSQL);
mapOCCI_->_put("HR.PERSON", &CPerson::readSQL, &CPerson::writeSQL);
mapOCCI_->_put("HR.STUDENT", &CStudent::readSQL, &CStudent::writeSQL);
}

Example 4–16  Listing of demo2.h for a Sample OCCI Application

#ifndef DEMO2_ORACLE
#define DEMO2_ORACLE

#ifndef OCCI_ORACLE
#include <occi.h>
#endif

using namespace std;
using namespace oracle::occi;

class MyFullName;
class MyAddress;
class MyPerson;
/* Changes ended here */

/* GENERATED DECLARATIONS FOR THE FULLNAME OBJECT TYPE. */
class CFullName : public oracle::occi::PObject {

private:
    OCCI_STD_NAMESPACE::string FIRST_NAME;
    OCCI_STD_NAMESPACE::string LAST_NAME;

public:
    OCCI_STD_NAMESPACE::string getFirst_name() const;
    void setFirst_name(const OCCI_STD_NAMESPACE::string &value);
    OCCI_STD_NAMESPACE::string getLast_name() const;
    void setLast_name(const OCCI_STD_NAMESPACE::string &value);
    void *operator new(size_t size);
    void operator new(size_t size, const oracle::occi::Connection * sess,
        const OCCI_STD_NAMESPACE::string &table);
    void *operator new(size_t size, void *ctxOCCI_);
    void *operator new(size_t size, const oracle::occi::Connection *sess,
        const OCCI_STD_NAMESPACE::string &tableName,
        const OCCI_STD_NAMESPACE::string &typeName,
        const OCCI_STD_NAMESPACE::string &tableSchema,
        const OCCI_STD_NAMESPACE::string &typeSchema);
    string getSQLTypeName() const;
    void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
        unsigned int &schemaNameLen, void **typeName,
        unsigned int &typeNameLen) const;
    CFullName();
    CFullName(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { }
    static void *readSQL(void *ctxOCCI_);
    virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
    static void writeSQL(void *objOCCI_, void *ctxOCCI_);
    virtual void writeSQL(oracle::occi::AnyData &streamOCCI_);
    ~CFullName();
};

/* GENERATED DECLARATIONS FOR THE ADDRESS OBJECT TYPE. */
class CAddress : public oracle::occi::PObject {

private:
    OCCI_STD_NAMESPACE::string STATE;
    OCCI_STD_NAMESPACE::string ZIP;

public:
    OCCI_STD_NAMESPACE::string getState() const;
    void setState(const OCCI_STD_NAMESPACE::string &value);
    OCCI_STD_NAMESPACE::string getZip() const;
    void setZip(const OCCI_STD_NAMESPACE::string &value);
    void *operator new(size_t size);
    void operator new(size_t size, const oracle::occi::Connection * sess,
        const OCCI_STD_NAMESPACE::string &table);
    void *operator new(size_t size, void *ctxOCCI_);
    void *operator new(size_t size, const oracle::occi::Connection *sess,
        const OCCI_STD_NAMESPACE::string &tableName,
        const OCCI_STD_NAMESPACE::string &typeName,
        const OCCI_STD_NAMESPACE::string &tableSchema,
        const OCCI_STD_NAMESPACE::string &typeSchema);
    string getSQLTypeName() const;
    void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
unsigned int &schemaNameLen, void **typeName,
unsigned int &typeNameLen) const;
CAddress();
CAddress(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
static void *readSQL(void *ctxOCCI_);
virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
static void writeSQL(void *objOCCI_, void *ctxOCCI_);
virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
-CAddress();
};

/* GENERATED DECLARATIONS FOR THE PERSON OBJECT TYPE. */
class CPerson : public oracle::occi::PObject {

private:
 oracle::occi::Number ID;
 MyFullName * NAME;
 oracle::occi::Ref< MyAddress > CURR_ADDR;
 OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > > PREV_ADDR_L;

public:
 oracle::occi::Number getId() const;
 void setId(const oracle::occi::Number &value);
 MyFullName * getName() const;
 void setName(MyFullName * value);
 oracle::occi::Ref< MyAddress > getCurr_addr() const;
 void setCurr_addr(const oracle::occi::Ref< MyAddress > &value);
 OCCI_STD_NAMESPACE::vector<oracle::occi::Ref< MyAddress>>>&
 getPrev_addr_l();
 const OCCI_STD_NAMESPACE::vector<oracle::occi::Ref<MyAddress>>&
 getPrev_addr_l() const;
 void setPrev_addr_l(const OCCI_STD_NAMESPACE::vector
 <oracle::occi::Ref< MyAddress>>>&
 getPrev_addr_l() const;
 void *operator new(size_t size);
 void *operator new(size_t size, const oracle::occi::Connection * sess,
 const OCCI_STD_NAMESPACE::string& table);
 void *operator new(size_t size, void *ctxOCCI_);
 void *operator new(size_t size, const oracle::occi::Connection * sess,
 const OCCI_STD_NAMESPACE::string &tableName,
 const OCCI_STD_NAMESPACE::string &typeName,
 const OCCI_STD_NAMESPACE::string &tableSchema,
 const OCCI_STD_NAMESPACE::string &typeSchema);;
string getSQLTypeName() const;
void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
 unsigned int &schemaNameLen, void **typeName,
 unsigned int &typeNameLen) const;
CPerson();
CPerson(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
static void *readSQL(void *ctxOCCI_);
virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
static void writeSQL(void *objOCCI_, void *ctxOCCI_);
virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
-CPerson();
};

/* GENERATED DECLARATIONS FOR THE STUDENT OBJECT TYPE. */
/* changes to the generated file - declarations for the MyPerson class. */
class MyPerson : public CPerson {

public:
A Sample OCCI Application

class CStudent : public MyPerson {
private:
    OCCI_STD_NAMESPACE::string SCHOOL_NAME;

public:
    OCCI_STD_NAMESPACE::string getSchool_name() const;
    void setSchool_name(const OCCI_STD_NAMESPACE::string &value);
    void *operator new(size_t size);
    void *operator new(size_t size, const oracle::occi::Connection * sess,
        const OCCI_STD_NAMESPACE::string& table);
    void *operator new(size_t size, const oracle::occi::Connection *sess,
        const OCCI_STD_NAMESPACE::string &tableName,
        const OCCI_STD_NAMESPACE::string &typeName,
        const OCCI_STD_NAMESPACE::string &tableSchema,
        const OCCI_STD_NAMESPACE::string &typeSchema);
    string getSQLTypeName() const;
    void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
        unsigned int &schemaNameLen, void **typeName,
        unsigned int &typeNameLen) const;
    CStudent();
    CStudent(void *ctxOCCI_) : MyPerson (ctxOCCI_) { }
    static void *readSQL(void *ctxOCCI_);
    virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
    static void writeSQL(void *objOCCI_, void *ctxOCCI_);
    virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
    ~CStudent();
};

/* changes made to the generated file */
/* declarations for the MyFullName class. */
class MyFullName : public CFullName {
public:
    MyFullName(string first_name, string last_name);
    void displayInfo();
    MyFullName(void *ctxOCCI_);
};

// declarations for the MyAddress class.
class MyAddress : public CAddress {
public:
    MyAddress(string state_i, string zip_i);
    void displayInfo();
    MyAddress(void *ctxOCCI_);
};

class MyStudent : public CStudent {
public:
    MyStudent(void *ctxOCCI_);
};
/* changes end here */
Example 4–17  Listing of demo2.cpp for a Sample OCCI Application

```cpp
#ifndef DEMO2_ORACLE
#include "demo2.h"
#endif

/* GENERATED METHOD IMPLEMENTATIONS FOR THE FULLNAME OBJECT TYPE. */
OCCI_STD_NAMESPACE::string CFullName::getFirst_name() const
{
    return FIRST_NAME;
}

void CFullName::setFirst_name(const OCCI_STD_NAMESPACE::string &value)
{
    FIRST_NAME = value;
}

OCCI_STD_NAMESPACE::string CFullName::getLast_name() const
{
    return LAST_NAME;
}

void CFullName::setLast_name(const OCCI_STD_NAMESPACE::string &value)
{
    LAST_NAME = value;
}

void *CFullName::operator new(size_t size)
{
    return oracle::occi::PObject::operator new(size);
}

void *CFullName::operator new(size_t size, const oracle::occi::Connection * sess,
    const OCCI_STD_NAMESPACE::string &table)
{
    return oracle::occi::PObject::operator new(size, sess, table,
        (char *) "HR.FULLNAME");
}

void *CFullName::operator new(size_t size, void *ctxOCCI_)
{
    return oracle::occi::PObject::operator new(size, ctxOCCI_);
}

void *CFullName::operator new(size_t size,
    const oracle::occi::Connection *sess,
    const OCCI_STD_NAMESPACE::string &tableName,
    const OCCI_STD_NAMESPACE::string &typeName,
    const OCCI_STD_NAMESPACE::string &tableSchema,
    const OCCI_STD_NAMESPACE::string &typeSchema)
{
    return oracle::occi::PObject::operator new(size, sess, tableName,
        typeName, tableSchema, typeSchema);
}

OCCI_STD_NAMESPACE::string CFullName::getSQLTypeName() const
{
    return OCCI_STD_NAMESPACE::string("HR.FULLNAME");
}
```

void CFullName::getSQLTypeName(oracle::occi::Environment *env,
    void **schemaName, unsigned int &schemaNameLen, void **typeName,
    unsigned int &typeNameLen) const
{
  PObject::getSQLTypeName(env, &CFullName::readSQL, schemaName,
    schemaNameLen, typeName, typeNameLen);
}

CFullName::CFullName()
{
}

void *CFullName::readSQL(void *ctxOCCI_)
{
  MyFullName *objOCCI_ = new(ctxOCCI_) MyFullName(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;  
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void CFullName::readSQL(oracle::occi::AnyData& streamOCCI_)
{
  FIRST_NAME = streamOCCI_.getString();
  LAST_NAME = streamOCCI_.getString();
}

void CFullName::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CFullName *objOCCI_ = (CFullName *) objectOCCI_;  
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}
void CFullName::writeSQL(oracle::occi::AnyData & streamOCCI_)
{
  streamOCCI_.setString(FIRST_NAME);
  streamOCCI_.setString(LAST_NAME);
}

CFullName::~CFullName()
{
  int i;
}

/* GENERATED METHOD IMPLEMENTATIONS FOR THE ADDRESS OBJECT TYPE. */
OCCI_STD_NAMESPACE::string CAddress::getState() const
{
  return STATE;
}

void CAddress::setState(const OCCI_STD_NAMESPACE::string & value)
{
  STATE = value;
}

OCCI_STD_NAMESPACE::string CAddress::getZip() const
{
  return ZIP;
}

void CAddress::setZip(const OCCI_STD_NAMESPACE::string & value)
{
  ZIP = value;
}

void *CAddress::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *CAddress::operator new(size_t size, const oracle::occi::Connection * sess,
const OCCI_STD_NAMESPACE::string & table)
{
  return oracle::occi::PObject::operator new(size, sess, (char *) "HR.ADDRESS");
}

void *CAddress::operator new(size_t size, void * ctxOCCI_)
{
  return oracle::occi::PObject::operator new(size, ctxOCCI_);
}

void *CAddress::operator new(size_t size, const oracle::occi::Connection * sess,
const OCCI_STD_NAMESPACE::string & tableName,
const OCCI_STD_NAMESPACE::string & typeName,
const OCCI_STD_NAMESPACE::string & tableSchema,
const OCCI_STD_NAMESPACE::string & typeSchema)
{
  return oracle::occi::PObject::operator new(size, sess, tableName,
    typeName, tableSchema, typeSchema);
}
OCCI_STD_NAMESPACE::string CAddress::getSQLTypeName() const
{
    return OCCI_STD_NAMESPACE::string("HR.ADDRESS");
}

void CAddress::getSQLTypeName(oracle::occi::Environment *env,
    void **schemaName,
    unsigned int &schemaNameLen,
    void **typeName,
    unsigned int &typeNameLen) const
{
    PObject::getSQLTypeName(env, &CAddress::readSQL, schemaName,
        schemaNameLen, typeName, typeNameLen);
}

CAddress::CAddress()
{
}

void *CAddress::readSQL(void *ctxOCCI_)
{
    MyAddress *objOCCI_ = new(ctxOCCI_) MyAddress(ctxOCCI_);
    oracle::occi::AnyData streamOCCI_(ctxOCCI_);

    try
    {
        if (streamOCCI_.isNull())
            objOCCI_->setNull();
        else
            objOCCI_->readSQL(streamOCCI_);
    }
    catch (oracle::occi::SQLException& excep)
    {
        delete objOCCI_;  
        excep.setErrorCtx(ctxOCCI_);  
        return (void *)NULL;
    }
    return (void *)objOCCI_;
}

void CAddress::readSQL(oracle::occi::AnyData& streamOCCI_)
{
    STATE = streamOCCI_.getString();
    ZIP = streamOCCI_.getString();
}

void CAddress::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
    CAddress *objOCCI_ = (CAddress *) objectOCCI_;
    oracle::occi::AnyData streamOCCI_(ctxOCCI_);

    try
    {
        if (objOCCI_->isNull())
            streamOCCI_.setNull();
        else
            objOCCI_->writeSQL(streamOCCI_);
    }
    catch (oracle::occi::SQLException& excep)
{  
    excep.setErrorCtx(ctxOCCI_); 
    return; 
}

void CAddress::writeSQL(oracle::occi::AnyData& streamOCCI_)
{  
    streamOCCI_.setString(STATE);  
    streamOCCI_.setString(ZIP); 
}

CAddress::~CAddress()
{  
    int i; 
}

/* GENERATED METHOD IMPLEMENTATIONS FOR THE PERSON OBJECT TYPE. */
oracle::occi::Number CPerson::getId() const
{  
    return ID; 
}

void CPerson::setId(const oracle::occi::Number &value)
{  
    ID = value; 
}

MyFullName * CPerson::getName() const
{  
    return NAME; 
}

void CPerson::setName(MyFullName * value)
{  
    NAME = value; 
}

oracle::occi::Ref< MyAddress > CPerson::getCurr_addr() const
{  
    return CURR_ADDR; 
}

void CPerson::setCurr_addr(const oracle::occi::Ref< MyAddress > &value)
{  
    CURR_ADDR = value; 
}

OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > >
& CPerson::getPrev_addr_l()
{  
    return PREV_ADDR_L; 
}

const OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > >
& CPerson::getPrev_addr_l() const
{  
    return PREV_ADDR_L; 
}
void CPerson::setPrev_addr_l(const OCCI_STD_NAMESPACE::vector<
    oracle::occi::Ref< MyAddress >> &value)
{
    PREV_ADDR_L = value;
}
void *CPerson::operator new(size_t size)
{
    return oracle::occi::PObject::operator new(size);
}
void *CPerson::operator new(size_t size,
    const oracle::occi::Connection * sess,
    const OCCI_STD_NAMESPACE::string& table)
{
    return oracle::occi::PObject::operator new(size, sess, 
        (char *) "HR.PERSON");
}
void *CPerson::operator new(size_t size, void *ctxOCCI_)
{
    return oracle::occi::PObject::operator new(size, ctxOCCI_);
}
void *CPerson::operator new(size_t size,
    const oracle::occi::Connection *sess,
    const OCCI_STD_NAMESPACE::string &tableName,
    const OCCI_STD_NAMESPACE::string &typeName,
    const OCCI_STD_NAMESPACE::string &tableSchema,
    const OCCI_STD_NAMESPACE::string &typeSchema)
{
    return oracle::occi::PObject::operator new(size, sess, 
        tableName, typeName, tableSchema, typeSchema);
}
OCCI_STD_NAMESPACE::string CPerson::getSQLTypeName() const
{
    return OCCI_STD_NAMESPACE::string("HR.PERSON");
}
void CPerson::getSQLTypeName(oracle::occi::Environment *env,
    void **schemaName,
    unsigned int &schemaNameLen,
    void **typeName,
    unsigned int &typeNameLen) const
{
    PObject::getSQLTypeName(env, &CPerson::readSQL, schemaName,
        schemaNameLen, typeName, typeNameLen);
}
CPerson::CPerson()
{
    NAME = (MyFullName *) 0;
}
void *CPerson::readSQL(void *ctxOCCI_)
{
    MyPerson *objOCCI_ = new(ctxOCCI_) MyPerson(ctxOCCI_);
    oracle::occi::AnyData streamOCCI_(ctxOCCI_);
    try
    {

if (streamOCCI_.isNull())
    objOCCI_->setNull();
else
    objOCCI_->readSQL(streamOCCI_);
}
catch (oracle::occi::SQLException& excep)
{
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)&NULL;
}
return (void *)objOCCI_;
}

void CPerson::readSQL(oracle::occi::AnyData& streamOCCI_)
{
    ID = streamOCCI_.getNumber();
    NAME = (MyFullName *) streamOCCI_.getObject(&MyFullName::readSQL);
    CURR_ADDR = streamOCCI_.getRef();
    oracle::occi::getVectorOfRefs(streamOCCI_, PREV_ADDR_L);
}

void CPerson::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
    streamOCCI_.setNumber(ID);
    streamOCCI_.setObject(NAME);
    streamOCCI_.setRef(CURR_ADDR);
    oracle::occi::setVectorOfRefs(streamOCCI_, PREV_ADDR_L);
}

CPerson::~CPerson()
{
    int i;
    delete NAME;
}

/* GENERATED METHOD IMPLEMENTATIONS FOR THE STUDENT OBJECT TYPE. */
OCCI_STD_NAMESPACE::string CStudent::getSchool_name() const
{
    return SCHOOL_NAME;
}
void CStudent::setSchool_name(const OCCI_STD_NAMESPACE::string &value)
{
    SCHOOL_NAME = value;
}

void *CStudent::operator new(size_t size)
{
    return oracle::occi::PObject::operator new(size);
}

void *CStudent::operator new(size_t size, const oracle::occi::Connection * sess,
    const OCCI_STD_NAMESPACE::string & table)
{
    return oracle::occi::PObject::operator new(size, sess, table,
        (char *) "HR.STUDENT");
}

void *CStudent::operator new(size_t size, void *ctxOCCI_)
{
    return oracle::occi::PObject::operator new(size, ctxOCCI_);
}

void *CStudent::operator new(size_t size, const oracle::occi::Connection *sess,
    const OCCI_STD_NAMESPACE::string &tableName,
    const OCCI_STD_NAMESPACE::string &typeName,
    const OCCI_STD_NAMESPACE::string &tableSchema,
    const OCCI_STD_NAMESPACE::string &typeSchema)
{
    return oracle::occi::PObject::operator new(size, sess, tableName,
        typeName, tableSchema, typeSchema);
}

OCCI_STD_NAMESPACE::string CStudent::getSQLTypeName() const
{
    return OCCI_STD_NAMESPACE::string("HR.STUDENT");
}

void CStudent::getSQLTypeName(oracle::occi::Environment *env,
    void **schemaName, unsigned int &schemaNameLen,
    void **typeName, unsigned int &typeNameLen) const
{
    PObject::getSQLTypeName(env, &CStudent::readSQL, schemaName,
        schemaNameLen, typeName, typeNameLen);
}

CStudent::CStudent()
{
}

void *CStudent::readSQL(void *ctxOCCI_)
{
    MyStudent *objOCCI_ = new(ctxOCCI_) MyStudent(ctxOCCI_);
    oracle::occi::AnyData streamOCCI_(ctxOCCI_);
    try
    {
        if (streamOCCI_.isNull())
Let us assume OTT generates FULL_NAME, ADDRESS, PERSON, and PFGFRCNT class declarations in demo2.h. The following sample OCCI application extends the classes generated by OTT, as specified in demo2.typ file in Example 4–12, and adds some user-defined methods. Note that these class declarations have been incorporated into demo2.h to ensure correct compilation.

Example 4–18  Listing of myDemo.h for a Sample OCCI Application

```c++
#ifndef MYDEMO_ORACLE
#define MYDEMO_ORACLE
#include <string>

void CStudent::readSQL(oracle::occi::AnyData& streamOCCI_)
{
    CPerson::readSQL(streamOCCI_);
    SCHOOL_NAME = streamOCCI_.getString();
}

void CStudent::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
    CPerson::writeSQL(streamOCCI_);
    streamOCCI_.setString(SCHOOL_NAME);
}

CStudent::~CStudent()
{
    int i;
}

Let us assume OTT generates FULL_NAME, ADDRESS, PERSON, and PFGFRCNT class declarations in demo2.h. The following sample OCCI application extends the classes generated by OTT, as specified in demo2.typ file in Example 4–12, and adds some user-defined methods. Note that these class declarations have been incorporated into demo2.h to ensure correct compilation.

Example 4–18  Listing of myDemo.h for a Sample OCCI Application

```
// declarations for the MyFullName class.
class MyFullName : public CFullName
{
    public:
    MyFullName(string first_name, string last_name);
    void displayInfo();
};

// declarations for the MyAddress class.
class MyAddress : public CAddress
{
    public:
    MyAddress(string state_i, string zip_i);
    void displayInfo();
};

// declarations for the MyPerson class.
class MyPerson : public CPerson
{
    public:
    MyPerson(Number id_i, MyFullname *name_i,
              const Ref<MyAddress>& addr_i);
    void move(const Ref<MyAddress>& new_addr);
    void displayInfo();
};

Example 4–19  Listing for myDemo.cpp for a Sample OCCI Application

/* initialize MyFullName */
MyFullName::MyFullName(string first_name,string last_name)
{
    setFirst_name(first_name);
    setLast_name(last_name);
}

/* display all the information in MyFullName */
void MyFullName::displayInfo()
{
    cout << 'FIRST NAME is' << getFirst_name() << endl;
    cout << 'LAST NAME is' << getLast_name() << endl;
}

MyFullName::MyFullName(void *ctxOCCI_):CFullName(ctxOCCI_)
{
}

/* METHOD IMPLEMENTATIONS FOR MyAddress CLASS. */
/* initialize MyAddress */
MyAddress::MyAddress(string state_i, string zip_i)
{
    setState(state_i);
    setZip(zip_i);
}

/* display all the information in MyAddress */
void MyAddress::displayInfo()
{
    cout << "STATE is" << getState() << endl;
    cout << "ZIP is" << getZip() << endl;
}

MyAddress::MyAddress(void *ctxOCCI_) :CAddress(ctxOCCI_)
{
}

/* METHOD IMPLEMENTATIONS FOR MyPerson CLASS. */

/* initialize MyPerson */
MyPerson::MyPerson(Number id_i, MyFullName* name_i,
const Ref<MyAddress>& addr_i)
{
    setId(id_i);
    setName(name_i);
    setCurr_addr(addr_i);
}

MyPerson::MyPerson(void *ctxOCCI_) :CPerson(ctxOCCI_)
{
}

/* move Person from curr_addr to new_addr */
void MyPerson::move(const Ref<MyAddress>& new_addr)
{
    // append curr_addr to the vector //
    getPrev_addr_l().push_back(getCurr_addr());
    setCurr_addr(new_addr);

    // mark the object as dirty
    this->markModified();
}

/* display all the information of MyPerson */
void MyPerson::displayInfo()
{
    cout << "ID is" << (int)getId() << endl;
    getName()->displayInfo();

    // de-referencing the Ref attribute using -> operator
    getCurr_addr() -> displayInfo();
    cout << "Prev Addr List: " << endl;
    for (int i = 0; i < getPrev_addr_l().size(); i++)
    {
        // access the collection elements using [] operator
        (getPrev_addr_l())[i] -> displayInfo();
    }
}
4-37

Example 4–20  Listing of main.cpp for a Sample OCCI Application

```cpp
#ifndef DEMO2_ORACLE
#include <demo2.h>
#endif

#ifndef MAPPINGS_ORACLE
#include <mappings.h>
#endif

#include <iostream>
using namespace std;
using namespace oracle;

int main()
{
    Environment *env = Environment::createEnvironment(Environment::OBJECT);
    mappings(env);

    try {
        Connection *conn = Connection("HR", "password");

        /* Call the OTT generated function to register the mappings */
        /* create a persistent object of type ADDRESS in the database table,
           ADDR_TAB */
        MyAddress *addr1 = new(conn, "ADDR_TAB") MyAddress("CA", "94065");
        conn->commit();

        Statement *st = conn->createStatement("select ref(a) from addr_tab a");
        ResultSet *rs = st->executeQuery();
        Ref<MyAddress> r1;
        if (rs->next())
        { r1 = rs->getRef(1); }
        st->closeResultSet(rs);
        conn->terminateStatement(st);

        MyFullName * name1 = new MyFullName("Joe", "Black");

        /* create a persistent object of type Person in the database table
           PERSON_TAB */
        MyPerson *person1 = new(conn, "PERSON_TAB") MyPerson(1,name1,r1);
        conn->commit();

        /* selecting the inserted information */
        Statement *stmt = conn->createStatement();
        ResultSet *resultSet =
            stmt->executeQuery("SELECT REF(a) from person_tab a where id = 1");

        if (resultSet->next())
        { Ref<MyPerson> joe_ref = (Ref<MyPerson>) resultSet->getRef(1);
```
joe_ref->displayInfo();

/* create a persistent object of type ADDRESS in the database table
   ADDR_TAB */
MyAddress *new_addr1 = new(conn, "ADDR_TAB") MyAddress("PA", "92140");
joe_ref->move(new_addr1->getRef());
joe_ref->displayInfo();
}

/* commit the transaction which results in the newly created object
   new_addr and the dirty object joe to be flushed to the server.
   Note that joe was marked dirty in move(). */
conn->commit();
conn->terminateStatement(stmt);
env->terminateConnection(conn);
}

catch ( exception &x)
{
    cout << x.what () << endl;
}
Environment::terminateEnvironment(env);
return 0;
}
This chapter is a reference for Oracle data types used by Oracle C++ Interface applications. This information helps you to understand the conversions between internal and external representations of data that occur when you transfer data between your application and the database server.

This chapter contains these topics:

- Overview of Oracle Data Types
- Internal Data Types
- External Data Types
- Data Conversions

### Overview of Oracle Data Types

Accurate communication between your C++ program and the Oracle database server is critical. OCCI applications can retrieve data from database tables by using SQL queries or they can modify existing data with SQL `INSERT`, `UPDATE`, and `DELETE` functions. To facilitate communication between the host language C++ and the database server, you must be aware of how C++ data types are converted to Oracle data types and back again.

In the Oracle database, values are stored in columns in tables. Internally, Oracle represents data in particular formats called internal data types. `NUMBER`, `VARCHAR2`, and `DATE` are examples of Oracle internal data types.

OCCI applications work with host language data types, or external data types, predefined by the host language. When data is transferred between an OCCI application and the database server, the data from the database is converted from internal data types to external data types.

### OCCI Type and Data Conversion

OCCI defines an enumerator called `Type` that lists the possible data representation formats available in an OCCI application. These representation formats are called external data types. When data is sent to the database server from the OCCI application, the external data type indicates to the database server what format to expect the data. When data is requested from the database server by the OCCI application, the external data type indicates the format of the data to be returned.

For example, on retrieving a value from a `NUMBER` column, the program may be set to retrieve it in `OCCIINT` format (a signed integer format into an integer variable). Or, the
client might be set to send data in OCCIFLOAT format (floating-point format) stored in a C++ float variable to be inserted in a column of NUMBER type.

An OCCI application binds input parameters to a Statement, by calling a setxxx() method (the external datatype is implicitly specified by the method name), or by calling the registerOutParam(), setDataBuffer(), or setDataBufferArray() method (the external data type is explicitly specified in the method call). Similarly, when data values are fetched through a ResultSet object, the external representation of the retrieved data must be specified. This is done by calling a getxxx() method (the external datatype is implicitly specified by the method name) or by calling the setDataBuffer() method (the external data type is explicitly specified in the method call).

Note that there are more external data types than internal data types. In some cases, a single external data type maps to a single internal data type; in other cases, many external data types map to a single internal data type. The many-to-one mapping provides you with added flexibility.

See Also: External Data Types on page 5-4

Internal Data Types

The internal (built-in) data types provided by Oracle are listed in this section. A brief summary of internal Oracle data types, including description, code, and maximum size, appears in Table 5-1.

Table 5-1 Summary of Oracle Internal Data Types

<table>
<thead>
<tr>
<th>Internal Data Type</th>
<th>Maximum Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFILE</td>
<td>4 gigabytes</td>
</tr>
<tr>
<td>BINARY_DOUBLE</td>
<td>8 bytes</td>
</tr>
<tr>
<td>BINARY_FLOAT</td>
<td>4 bytes</td>
</tr>
<tr>
<td>CHAR</td>
<td>2,000 bytes</td>
</tr>
<tr>
<td>DATE</td>
<td>7 bytes</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND REF</td>
<td>11 bytes</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH REF</td>
<td>5 bytes</td>
</tr>
<tr>
<td>LONG</td>
<td>2 gigabytes (2^31-1 bytes)</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>2 gigabytes (2^31-1 bytes)</td>
</tr>
<tr>
<td>NCHAR</td>
<td>2,000 bytes</td>
</tr>
<tr>
<td>NUMBER</td>
<td>21 bytes</td>
</tr>
<tr>
<td>NVARCHAR2</td>
<td>4,000 bytes</td>
</tr>
<tr>
<td>RAW</td>
<td>2,000 bytes</td>
</tr>
<tr>
<td>REF</td>
<td></td>
</tr>
<tr>
<td>BLOB</td>
<td>4 gigabytes</td>
</tr>
<tr>
<td>CLOB</td>
<td>4 gigabytes</td>
</tr>
<tr>
<td>NCLOB</td>
<td>4 gigabytes</td>
</tr>
<tr>
<td>ROWID</td>
<td>10 bytes</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>11 bytes</td>
</tr>
</tbody>
</table>
You can use five Oracle internal data types to specify columns that contain either characters or arrays of bytes: **CHAR**, **VARCHAR2**, **RAW**, **LONG**, and **LONG RAW**.

**CHAR**, **VARCHAR2**, and **LONG** columns normally hold character data. **RAW** and **LONG RAW** hold bytes that are not interpreted as characters, for example, pixel values in a bitmapped graphics image. Character data can be transformed when passed through a gateway between networks. For example, character data passed between systems by using different languages (where single characters may be represented by differing numbers of bytes) can be significantly changed in length. Raw data is never converted in this way.

The database designer is responsible for choosing the appropriate Oracle internal data type for each column in a table. You must be aware of the many possible ways that character and byte-array data can be represented and converted between variables in the OCCI program and Oracle database tables.

### Universal Rowid (UROWID)

The universal rowid (**UROWID**) is a data type that can store both the logical and the physical rowid of rows in Oracle tables and in foreign tables, such as DB2 tables accessed through a gateway. Logical rowid values are primary key-based logical identifiers for the rows of index-organized tables.

To use columns of the **UROWID** data type, the value of the **COMPATIBLE** initialization parameter must be set to 8.1 or higher.

The following **OCCI_SQLT** types can be bound to universal rowids:

- **OCCI_SQLT_CHR** (**VARCHAR2**)
- **OCCI_SQLT_VCS** (**VARCHAR**)
- **OCCI_SQLT_STR** (**NULL terminated string**)
- **OCCI_SQLT_LVC** (**long VARCHAR**)
- **OCCI_SQLT_AFC** (**CHAR**)
- **OCCI_SQLT_AVVC** (**CHARZ**)
- **OCCI_SQLT_VST** (**string**)

### See Also:
- *Oracle Database SQL Language Reference*
- *Oracle Database Concepts*
External Data Types

OCCI application communicate with the Oracle database server by using external data types. Specifically, external data types are mapped to C++ data types.

Table 5–2 lists the Oracle external data types, the C++ equivalent (what the Oracle internal data type is usually converted to), and the corresponding OCCI type. Note the following conditions:

- In C++ Data Type column, n stands for variable length and depends on program requirements or operating system.
- The usage of types in Statement class methods is as follows:
  - setDataBuffer() and setDataBufferArray(): Only types of the form OCCI_SQLT_XXX (for example, OCCI_SQLT_INT) in the occiCommon.h file are permitted.
  - registerOutParam(): Only types of the form OCCIXXX (for example, OCCIDOUBLE, OCCICURSOR, and so on) on the occiCommon.h file are permitted. However, there are some exceptions: OCCIANYDATA, OCCIMETADATA, OCCISTREAM, and OCCIBOOL are not permitted.
- In the ResultSet class, only types of the form OCCI_SQLT_XXX (for example, OCCI_SQLT_INT) in the occiCommon.h file are permitted for use in setDataBuffer() and setDataBufferArray() methods.
- The TIMESTAMP and TIMESTAMPT WITH TIME ZONE data types are collectively known as datetimes. The INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND are collectively known as intervals.

<table>
<thead>
<tr>
<th>External Data Type</th>
<th>C++ Type</th>
<th>OCCI Type</th>
<th>Usage Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit signed</td>
<td>signed short,</td>
<td>OCCIINT</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>INTEGER</td>
<td>signed int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 bit signed</td>
<td>signed int,</td>
<td>OCCIINT</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>INTEGER</td>
<td>signed long</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 bit signed</td>
<td>signed char</td>
<td>OCCIINT</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFILE</td>
<td>Bfile</td>
<td>OCCIBFILE</td>
<td>Use with registerOutParam().</td>
</tr>
<tr>
<td>FFILE</td>
<td>OCILobLocator</td>
<td>OCCI_SQLT_FILE</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>BLOB</td>
<td>OCILobLocator</td>
<td>OCCI_SQLT_BLOB</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>BLOB</td>
<td>Blob</td>
<td>OCCIBLOB</td>
<td>Use with registerOutParam().</td>
</tr>
<tr>
<td>BOOL</td>
<td>bool</td>
<td>OCCIBOOL</td>
<td>Use with registerOutParam().</td>
</tr>
<tr>
<td>BYTES</td>
<td>Bytes</td>
<td>OCCIBYTES</td>
<td>Use with registerOutParam().</td>
</tr>
<tr>
<td>CHAR</td>
<td>char[n]</td>
<td>OCCI_SQLT_AFC</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>CHAR</td>
<td>string</td>
<td>OCCICHAR</td>
<td>Use with registerOutParam().</td>
</tr>
<tr>
<td>CLOB</td>
<td>OCILobLocator</td>
<td>OCCI_SQLT_CLOB</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
</tbody>
</table>

- OCCI_SQLT_RDD (ROWID descriptor)
## Table 5–2 (Cont.) External Data Types and Corresponding C++ and OCCI Types

<table>
<thead>
<tr>
<th>External Data Type</th>
<th>C++ Type</th>
<th>OCCI Type</th>
<th>Usage Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARZ</td>
<td>char[n+1]</td>
<td>OCCI_SQLT_RDD</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>CLOB</td>
<td>Clob</td>
<td>OCCICLOB</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>CURSOR</td>
<td>ResultSet</td>
<td>OCCICURSOR</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>DATE</td>
<td>char[7]</td>
<td>OCCI_SQLT_DAT</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>DATE</td>
<td>Date</td>
<td>OCCIDATE</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
<td>OCCIDOUBLE</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float, double</td>
<td>OCCIFLOAT</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
<td>OCCIFLOAT</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
<td>OCCIINT</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td>char[11]</td>
<td>OCCI_SQLT_INTE RVAL_DS</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>char[5]</td>
<td>OCCI_SQLT_INTE RVAL_YM</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>INTERVALDS</td>
<td>IntervalDS</td>
<td>OCCIINTERVALDS</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>INTERVALYM</td>
<td>IntervalYM</td>
<td>OCCIINTERVALYM</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>LONG</td>
<td>char[n]</td>
<td>OCCI_SQLT_LNG</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>unsigned char[n]</td>
<td>OCCI_SQLT_LBI</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>char[n+sizeof(integer)]</td>
<td>OCCI_SQLT_LVC</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>unsigned char[n+sizeof(integer)]</td>
<td>OCCI_SQLT_LVB</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>METADATA</td>
<td>MetaData</td>
<td>OCCIMETADATA</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>NAMED DATA TYPE</td>
<td>struct</td>
<td>OCCI_SQLT_NTY</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>NATIVE DOUBLE</td>
<td>double</td>
<td>OCCIBDOUBLE</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>NATIVE DOUBLE</td>
<td>Bdouble, double</td>
<td>OCCIBDOUBLE</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>NATIVE FLOAT</td>
<td>float</td>
<td>OCCIBFLOAT</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>NATIVE FLOAT</td>
<td>BFloat, float</td>
<td>OCCIBFLOAT</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>null terminated STRING</td>
<td>char[n+1]</td>
<td>OCCI_SQLT_STR</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>NUMBER</td>
<td>unsigned char[21]</td>
<td>OCCI_SQLT_NUM</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Number</td>
<td>OCCINUMBER</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>OBJECT</td>
<td>User defined types generated by OTT utility.</td>
<td>OCCIPOBJECT</td>
<td>Use with regiserOutParam().</td>
</tr>
</tbody>
</table>
### Description of External Data Types

This section provides a description for each of the external data types.

**BFILE**

The external data type BFILE allows read-only byte stream access to large files on the file system of the database server. A BFILE is a large binary data object stored in operating system files outside database tablespaces. These files use reference semantics. The Oracle server can access a BFILE provided the underlying server operating system supports stream-mode access to these operating system files.

<table>
<thead>
<tr>
<th>External Data Type</th>
<th>C++ Type</th>
<th>OCCI Type</th>
<th>Usage Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>unsigned char[n]</td>
<td>OCCI_SQLT_BIN</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>REF</td>
<td>OCIRef</td>
<td>OCCI_SQLT_REF</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>REF</td>
<td>Ref</td>
<td>OCCIREF</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>REFANY</td>
<td>RefAny</td>
<td>OCCIREFANY</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>ROWID</td>
<td>OCIRowid</td>
<td>OCCI_SQLT_RID</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>ROWID descriptor</td>
<td>Bytes</td>
<td>OCCIROWID</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>STRING</td>
<td>STL string</td>
<td>OCCISTRING</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>char[11]</td>
<td>OCCI_SQLT_TIME_STAMP</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Timestamp</td>
<td>OCCITIMESTAMP</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>TIMESTAMP WITH LOCAL TIME ZONE</td>
<td>char[7]</td>
<td>OCCI_SQLT_TIME_STAMP_LTZ</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>char[13]</td>
<td>OCCI_SQLT_TIME_STAMP_TZ</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>UNSIGNED INT</td>
<td>unsigned int</td>
<td>OCCIUNSIGNED_INT</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>UNSIGNED INT</td>
<td>unsigned int</td>
<td>OCCIUNSIGNED_INT</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>char[n+sizeof(short integer)]</td>
<td>OCCI_SQLT_VCS</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>char[n]</td>
<td>OCCI_SQLT_CHR</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>VARNUM</td>
<td>char[22]</td>
<td>OCCI_SQLT_VNU</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>VARRAW</td>
<td>unsigned char[n+sizeof(short integer)]</td>
<td>OCCI_SQLT_VBI</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>VECTOR</td>
<td>STL vector</td>
<td>OCCIVECTOR</td>
<td>Use with regiserOutParam().</td>
</tr>
</tbody>
</table>

---

**Table 5–2 (Cont.) External Data Types and Corresponding C++ and OCCI Types**

<table>
<thead>
<tr>
<th>External Data Type</th>
<th>C++ Type</th>
<th>OCCI Type</th>
<th>Usage Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>unsigned char[n]</td>
<td>OCCI_SQLT_BIN</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>REF</td>
<td>OCIRef</td>
<td>OCCI_SQLT_REF</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>REF</td>
<td>Ref</td>
<td>OCCIREF</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>REFANY</td>
<td>RefAny</td>
<td>OCCIREFANY</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>ROWID</td>
<td>OCIRowid</td>
<td>OCCI_SQLT_RID</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>ROWID descriptor</td>
<td>Bytes</td>
<td>OCCIROWID</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>STRING</td>
<td>STL string</td>
<td>OCCISTRING</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>char[11]</td>
<td>OCCI_SQLT_TIME_STAMP</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Timestamp</td>
<td>OCCITIMESTAMP</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>TIMESTAMP WITH LOCAL TIME ZONE</td>
<td>char[7]</td>
<td>OCCI_SQLT_TIME_STAMP_LTZ</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>char[13]</td>
<td>OCCI_SQLT_TIME_STAMP_TZ</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>UNSIGNED INT</td>
<td>unsigned int</td>
<td>OCCIUNSIGNED_INT</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>UNSIGNED INT</td>
<td>unsigned int</td>
<td>OCCIUNSIGNED_INT</td>
<td>Use with regiserOutParam().</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>char[n+sizeof(short integer)]</td>
<td>OCCI_SQLT_VCS</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>char[n]</td>
<td>OCCI_SQLT_CHR</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>VARNUM</td>
<td>char[22]</td>
<td>OCCI_SQLT_VNU</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>VARRAW</td>
<td>unsigned char[n+sizeof(short integer)]</td>
<td>OCCI_SQLT_VBI</td>
<td>Use with setDataBuffer(), setDataBufferArray().</td>
</tr>
<tr>
<td>VECTOR</td>
<td>STL vector</td>
<td>OCCIVECTOR</td>
<td>Use with regiserOutParam().</td>
</tr>
</tbody>
</table>
**BDOUBLE**

The **BDouble** interface in OCCI encapsulates the native double data and the **NULL** information of a column or object attribute of the type `binary_double`. The OCCI methods in AnyData Class, ResultSet Class and Statement Class, and the global methods that take these class objects as parameters, use the following definition for the `BDOUBLE` data type:

**Example 5–1  Definition of the BDOUBLE Data Type**

```c
struct BDouble
{
    double value;
    bool isNull;

    BDouble()
    {
        isNull = false;
        value = 0.;
    }
};
```

**BFLOAT**

The **BFloat** interface in OCCI encapsulates the native float data and the **NULL** information of a column or object attribute of the type `binary_float`. The OCCI methods in AnyData Class, ResultSet Class and Statement Class, and the global methods that take these class objects as parameters, use the following definition for the `BFLOAT` data type:

**Example 5–2  Definition of the BFLOAT Data Type**

```c
struct BFloat
{
    float value;
    bool isNull;

    BFloat()
    {
        isNull = false;
        value = 0.;
    }
};
```

**BLOB**

The external data type **BLOB** stores unstructured binary large objects. A BLOB can be thought of as a bitstream with no character set semantics. BLOBs can store up to 4 gigabytes of binary data.

BLOB data types have full transactional support. Changes made through OCCI participate fully in the transaction. BLOB value manipulations can be committed or rolled back. You cannot save a BLOB locator in a variable in one transaction and then use it in another transaction or session.

**CHAR**

The external data type **CHAR** is a string of characters, with a maximum length of 2000 characters. Character strings are compared by using blank-padded comparison semantics.
CHARZ
The external data type CHARZ is similar to the CHAR data type, except that the string must be NULL terminated on input, and Oracle places a NULL terminator character at the end of the string on output. The NULL terminator serves only to delimit the string on input or output. It is not part of the data in the table.

CLOB
The external data type CLOB stores fixed-width or varying-width character data. A CLOB can store up to 4 gigabytes of character data. CLOBs have full transactional support. Changes made through OCCI participate fully in the transaction. CLOB value manipulations can be committed or rolled back. You cannot save a CLOB locator in a variable in one transaction and then use it in another transaction or session.

DATE
The external data type DATE can update, insert, or retrieve a date value using the Oracle internal seven byte date binary format, as listed in Table 5–3:

Table 5–3 Format of the DATE Data Type

<table>
<thead>
<tr>
<th>Example</th>
<th>Byte 1: Century</th>
<th>Byte 2: Year</th>
<th>Byte 3: Month</th>
<th>Byte 4: Day</th>
<th>Byte 5: Hour</th>
<th>Byte 6: Minute</th>
<th>Byte 7: Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: 01-JUN-2000, 3:17PM</td>
<td>120</td>
<td>100</td>
<td>6</td>
<td>1</td>
<td>16</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>2: 01-JAN-4712 BCE</td>
<td>53</td>
<td>88</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Example 1, 01-JUN-2000, 3:17PM:
- The century and year bytes (1 and 2) are in excess-100 notation. Dates BCE (Before Common Era) are less than 100. Dates in the Common Era (CE), 0 and after, are greater than 100. For dates 0 and after, the first digit of both bytes 1 and 2 signifies that it is of the CE.
- For byte 1, the second and third digits of the century are calculated as the year (an integer) divided by 100. With integer division, the fractional portion is discarded. The following calculation is for the year 1992: 1992 / 100 = 19.
- For byte 1, 119 represents the twentieth century, 1900 to 1999. A value of 120 would represent the twenty-first century, 2000 to 2099.
- For byte 2, the second and third digits of the year are calculated as the year modulo 100: 1992 % 100 = 92.
- For byte 2, 192 represents the ninety-second year of the current century. A value of 100 would represent the zeroth year of the current century.
- The year 2000 would yield 120 for byte 1 and 100 for byte 2.
- For bytes 3 through 7, valid dates begin at 01-JAN of the year. The month byte ranges from 1 to 12, the date byte ranges from 1 to 31, the hour byte ranges from 1 to 24, the minute byte ranges from 1 to 60, and the second byte ranges from 1 to 60.

Example 2, 01-JAN-4712 BCE:
- For years before 0 CE, centuries and years are represented by the difference between 100 and the number.
- For byte 1, 01-JAN-4712 BCE is century 53: 100 - 47 = 53.
For byte 2, 01-JAN-4712 BCE is year 88: 100 - 12 = 88.

If no time is specified for a date, the time defaults to midnight and bytes 5 through 6 are set to 1:1,1,1.

When you enter a date in binary format by using the external data type DATE, the database does not perform consistency or range checking. All data in this format must be validated before input.

There is little need for the external data type DATE. It is more convenient to convert DATE values to a character format, because most programs deal with dates in a character format, such as DD-MON-YYYY. Instead, you may use the Date data type.

When a DATE column is converted to a character string in your program, it is returned in the default format mask for your session, or as specified in the INIT.ORA file.

This data type is different from OCCI DATE which corresponds to a C++ Date data type.

FLOAT
The external data type FLOAT processes numbers with fractional parts. The number is represented in the host system’s floating-point format. Normally, the length is 4 or 8 bytes.

The internal format of an Oracle number is decimal. Most floating-point implementations are binary. Oracle, therefore, represents numbers with greater precision than floating-point representations.

INTEGER
The external data type INTEGER is used for converting numbers. An external integer is a signed binary number. Its size is operating system-dependent. If the number being returned from Oracle is not an integer, then the fractional part is discarded, and no error is returned. If the number returned exceeds the capacity of a signed integer for the system, then Oracle returns an overflow on conversion error.

A rounding error may occur when converting between FLOAT and NUMBER. Using a FLOAT as a bind variable in a query may return an error. You can work around this by converting the FLOAT to a string and using the OCCI type OCCI_SQLT_CHR or the OCCI type OCCI_SQLT_STR for the operation.

INTERVAL DAY TO SECOND
The external data type INTERVAL DAY TO SECOND stores the difference between two datetime values in terms of days, hours, minutes, and seconds. Specify this data type as follows:

```
INTERVAL DAY [(day_precision)]
    TO SECOND [(fractional_seconds_precision)]
```

This example uses the following placeholders:

- **day_precision**: Number of digits in the DAY datetime field. Accepted values are 1 to 9. The default is 2.
- **fractional_seconds_precision**: Number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.

To specify an INTERVAL DAY TO SECOND literal with nondefault day and second precision, you must specify the precisions in the literal. For example, you might
specify an interval of 100 days, 10 hours, 20 minutes, 42 seconds, and 22 hundredths of a second as follows:

```
INTERVAL '100 10:20:42.22' DAY(3) TO SECOND(2)
```

You can also use abbreviated forms of the `INTERVAL` `DAY` TO `SECOND` literal. For example:

- `INTERVAL '90' MINUTE maps to INTERVAL '00 00:90:00.00' DAY TO SECOND(2)`
- `INTERVAL '30:30' HOUR TO MINUTE maps to INTERVAL '00 30:30:00.00' DAY TO SECOND(2)`
- `INTERVAL '30' SECOND(2,2) maps to INTERVAL '00 00:00:30.00' DAY TO SECOND(2)`

**INTERVAL YEAR TO MONTH**

The external data type `INTERVAL` `YEAR` TO `MONTH` stores the difference between two datetime values by using the `YEAR` and `MONTH` datetime fields. Specify `INTERVAL` `YEAR` TO `MONTH` as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

The placeholder `year_precision` is the number of digits in the `YEAR` datetime field. The default value of `year_precision` is 2. To specify an `INTERVAL` `YEAR` TO `MONTH` literal with a nondefault `year_precision`, you must specify the precision in the literal. For example, the following `INTERVAL` `YEAR` TO `MONTH` literal indicates an interval of 123 years, 2 months:

```
INTERVAL '123-2' YEAR(3) TO MONTH
```

You can also use abbreviated forms of the `INTERVAL` `YEAR` TO `MONTH` literal. For example,

- `INTERVAL '10' MONTH maps to INTERVAL '0-10' YEAR TO MONTH`
- `INTERVAL '123' YEAR(3) maps to INTERVAL '123-0' YEAR(3) TO MONTH`

**LONG**

The external data type `LONG` stores character strings longer than 4000 bytes and up to 2 gigabytes in a column of data type `LONG`. Columns of this type are only used for storage and retrieval of long strings. They cannot be used in methods, expressions, or `WHERE` clauses. `LONG` column values are generally converted to and from character strings.

**LONG RAW**

The external data type `LONG RAW` is similar to the external data type `RAW`, except that it stores up to 2 gigabytes.

**LONG VARCHAR**

The external data type `LONG VARCHAR` stores data from and into an Oracle `LONG` column. The first four bytes contain the length of the item. The maximum length of a `LONG VARCHAR` is 2 gigabytes.
LONG VARRAW
The external data type LONG VARRAW store data from and into an Oracle LONG RAW column. The length is contained in the first four bytes. The maximum length is 2 gigabytes.

NCLOB
The external data type NCLOB is a national character version of a CLOB. It stores fixed-width, multibyte national character set character (NCHAR), or varying-width character set data. An NCLOB can store up to 4 gigabytes of character text data.

NCLOBs have full transactional support. Changes made through OCCI participate fully in the transaction. NCLOB value manipulations can be committed or rolled back. You cannot save an NCLOB locator in a variable in one transaction and then use it in another transaction or session.

You cannot create an object with NCLOB attributes, but you can specify NCLOB parameters in methods.

NUMBER
You should not have to use NUMBER as an external data type. If you do use it, Oracle returns numeric values in its internal 21-byte binary format and expects this format on input. The following discussion is included for completeness only.

Oracle stores values of the NUMBER data type in a variable-length format. The first byte is the exponent and is followed by 1 to 20 mantissa bytes. The high-order bit of the exponent byte is the sign bit; it is set for positive numbers and it is cleared for negative numbers. The lower 7 bits represent the exponent, which is a base-100 digit with an offset of 65.

To calculate the decimal exponent, add 65 to the base-100 exponent and add another 128 if the number is positive. If the number is negative, you do the same, but subsequently the bits are inverted. For example, -5 has a base-100 exponent = 62 (0x3e). The decimal exponent is thus (-0x3e)-128-65 = 0xc1-128-65 = 193-128-65 = 0.

Each mantissa byte is a base-100 digit, in the range 1 to 100. For positive numbers, the digit has 1 added to it. So, the mantissa digit for the value 5 is 6. For negative numbers, instead of adding 1, the digit is subtracted from 101. So, the mantissa digit for the number -5 is: 101-5 = 96. Negative numbers have a byte containing 102 appended to the data bytes. However, negative numbers that have 20 mantissa bytes do not have the trailing 102 byte. Because the mantissa digits are stored in base-100, each byte can represent two decimal digits. The mantissa is normalized; leading zeroes are not stored.

Up to 20 data bytes can represent the mantissa. However, only 19 are guaranteed to be accurate. The 19 data bytes, each representing a base-100 digit, yield a maximum precision of 38 digits for an internal data type NUMBER.

Note that this data type is different from OCCI NUMBER which corresponds to a C++ Number data type.

OCCI BFILE

See Also: Chapter 13, "OCCI Application Programming Interface", Bfile Class on page 13-20
OCCI BLOB

See Also: Chapter 13, "OCCI Application Programming Interface", Blob Class on page 13-26

OCCI BYTES

See Also: Chapter 13, "OCCI Application Programming Interface", Bytes Class on page 13-35

OCCI CLOB

See Also: Chapter 13, "OCCI Application Programming Interface", Clob Class on page 13-38

OCCI DATE

See Also: Chapter 13, "OCCI Application Programming Interface", Date Class on page 13-75

OCCI INTERVALDS

See Also: Chapter 13, "OCCI Application Programming Interface", IntervalDS Class on page 13-97

OCCI INTERVALYM

See Also: Chapter 13, "OCCI Application Programming Interface", IntervalYM Class on page 13-107

OCCI NUMBER

See Also: Chapter 13, "OCCI Application Programming Interface", Number Class on page 13-143

OCCI POBJECT

See Also: Chapter 13, "OCCI Application Programming Interface", PObject Class on page 13-162

OCCI REF

See Also: Chapter 13, "OCCI Application Programming Interface", Ref Class on page 13-173

OCCI REFANY

See Also: Chapter 13, "OCCI Application Programming Interface", RefAny Class on page 13-178

OCCI STRING

The external data type OCCI STRING corresponds to an STL string.
OCCI TIMESTAMP

See Also: Chapter 13, "OCCI Application Programming Interface", Timestamp Class on page 13-264

OCCI VECTOR
The external data type OCCI VECTOR is used to represent collections, for example, a nested table or VARRAY. CREATE TYPE num_type as VARRAY OF NUMBER(10) can be represented in a C++ application as vector<int>, vector<Number>, and so on.

RAW
The external data type RAW is used for binary data or byte strings that are not to be interpreted or processed by Oracle. RAW could be used, for example, for graphics character sequences. The maximum length of a RAW column is 2000 bytes.

When RAW data in an Oracle table is converted to a character string, the data is represented in hexadecimal code. Each byte of RAW data is represented as two characters that indicate the value of the byte, ranging from 00 to FF. If you input a character string by using RAW, then you must use hexadecimal coding.

REF
The external data type REF is a reference to a named data type. To allocate a REF for use in an application, declare a variable as a pointer to a REF.

ROWID
The external data type ROWID identifies a particular row in a database table. The ROWID is often returned from a query by issuing a statement similar to the following example:

SELECT ROWID, var1, var2 FROM db;

You can then use the returned ROWID in further DELETE statements.

If you are performing a SELECT for an UPDATE operation, then the ROWID is implicitly returned.

STRING
The external data type STRING behaves like the external data type VARCHAR2 (data type code 1), except that the external data type STRING must be NULL-terminated.

Note that this data type is different from OCCI STRING which corresponds to a C++ STL string data type.

TIMESTAMP
The external data type TIMESTAMP is an extension of the DATE data type. It stores the year, month, and day of the DATE data type, plus hour, minute, and second values. Specify the TIMESTAMP data type as follows:

TIMESTAMP [(fractional_seconds_precision)]

The placeholder fractional_seconds_precision optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6. For example, you specify TIMESTAMP(2) as a literal as follows:
**TIMESTAMP '1997-01-31 09:26:50.10'

Note that this data type is different from OCCI TIMESTAMP.

**TIMESTAMP WITH LOCAL TIME ZONE**

The external data type `TIMESTAMP WITH TIME ZONE` (TSTZ) is a variant of `TIMESTAMP` that includes an explicit time zone displacement in its value. The time zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly Greenwich Mean Time. Specify the `TIMESTAMP WITH TIME ZONE` data type as follows:

`TIMESTAMP(fractional_seconds_precision) WITH TIME ZONE`

The placeholder `fractional_seconds_precision` optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

Two `TIMESTAMP WITH TIME ZONE` values are considered identical if they represent the same instant in UTC, regardless of the `TIME ZONE` offsets stored in the data.

**UNSIGNED INT**

The external data type `UNSIGNED INT` is used for unsigned binary integers. The size in bytes is operating system dependent. The host system architecture determines the order of the bytes in a word. If the number being output from Oracle is not an integer, the fractional part is discarded, and no error is returned. If the number to be returned exceeds the capacity of an unsigned integer for the operating system, Oracle returns an overflow on conversion error.

**VARCHAR**

The external data type `VARCHAR` store character strings of varying length. The first two bytes contain the length of the character string, and the remaining bytes contain the actual string. The specified length of the string in a bind or a define call must include the two length bytes, meaning the largest `VARCHAR` string is 65533 bytes long, not 65535. For converting longer strings, use the `LONG VARCHAR` external data type.

**VARCHAR2**

The external data type `VARCHAR2` is a variable-length string of characters up to 4000 bytes.
**VARNUM**

The external data type *VARNUM* is similar to the external data type *NUMBER*, except that the first byte contains the length of the number representation. This length value does not include the length byte itself. Reserve 22 bytes to receive the longest possible *VARNUM*. You must set the length byte when you send a *VARNUM* value to the database.

**Table 5-4  VARNUM Examples**

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Length Byte</th>
<th>Exponent Byte</th>
<th>Mantissa Bytes</th>
<th>Terminator Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>128</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>193</td>
<td>6</td>
<td>N/A</td>
</tr>
<tr>
<td>-5</td>
<td>3</td>
<td>62</td>
<td>96</td>
<td>102</td>
</tr>
<tr>
<td>2767</td>
<td>3</td>
<td>194</td>
<td>28, 68</td>
<td>N/A</td>
</tr>
<tr>
<td>-2767</td>
<td>4</td>
<td>61</td>
<td>74, 34</td>
<td>102</td>
</tr>
<tr>
<td>100000</td>
<td>2</td>
<td>195</td>
<td>11</td>
<td>N/A</td>
</tr>
<tr>
<td>1234567</td>
<td>5</td>
<td>196</td>
<td>2, 24, 46, 68</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**VARRAW**

The external data type *VARRAW* is similar to the external data type *RAW*, except that the first two bytes contain the length of the data. The specified length of the string in a bind or a define call must include the two length bytes. So the largest *VARRAW* string that can be received or sent is 65533 bytes, not 65535. For converting longer strings, use the *LONG VARRAW* data type.

**NATIVE DOUBLE**

This external data type implements the IEEE 754 standard double-precision floating point data type. It is represented in the host system’s native floating point format. The data type is stored in the Oracle Server in a byte comparable canonical format, and requires 8 bytes for storage, including the length byte. It is an alternative to Oracle *NUMBER* and has the following advantages over *NUMBER*:

- Fewer bytes used in storage
- Matches data types used by RDBMS Clients
- Supports a wider range of values used in scientific calculations.

**NATIVE FLOAT**

This external data type implements the IEEE 754 single-precision floating point data type. It is represented in the host system’s native floating point format. The data type is stored in the Oracle Server in a byte comparable canonical format, and requires 4 bytes for storage, including the length byte. It is an alternative to Oracle *NUMBER* and has the following advantages over *NUMBER*:

- Fewer bytes used in storage
- Matches data types used by RDBMS Clients
- Supports a wider range of values used in scientific calculations
Data Conversions

Table 5–5 lists the supported conversions from Oracle internal data types to external data types, and from external data types to internal column representations.

Note the following conditions:

- A REF stored in the database is converted to OCCI_SQLT_REF on output.
- An OCCI_SQLT_REF is converted to the internal representation of a REF on input.
- A named data type stored in the database is converted to OCCI_SQLT_NTY (and represented by a C structure in the application) on output.
- An OCCI_SQLT_NTY (represented by a C structure in an application) is converted to the internal representation of the corresponding data type on input.
- LOBs and BFILES are represented by descriptors in OCCI applications, so there are no input or output conversions.

Also note that in Table 5–5, conversions have the following numeric codes:

1. The data type must be in Oracle ROWID format for input; it is returned in Oracle ROWID format on output.
2. The data type must be in Oracle DATE format for input; it is returned in Oracle DATE format on output.
3. The data type must be in hexadecimal format for input; it is returned in hexadecimal format on output.
4. The data type must represent a valid number for output.
5. The length must be less than or equal to 2000 characters.
6. The data type must be stored in hexadecimal format on output; it is in hexadecimal format on output.

### Table 5–5 Data Conversions Between External and Internal Data Types

<table>
<thead>
<tr>
<th>External Data Types</th>
<th>Internal Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR 2</td>
<td>NUMBER</td>
</tr>
<tr>
<td>CHAR</td>
<td>I/O</td>
</tr>
<tr>
<td>CHARZ</td>
<td>I/O</td>
</tr>
<tr>
<td>DATE</td>
<td>I/O</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>I/O^4</td>
</tr>
<tr>
<td>FLOAT</td>
<td>I/O^4</td>
</tr>
<tr>
<td>INTEGER</td>
<td>I/O^4</td>
</tr>
<tr>
<td>LONG</td>
<td>I/O</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>O^6</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>I/O</td>
</tr>
<tr>
<td>LONG VARRAW</td>
<td>I/O^6</td>
</tr>
<tr>
<td>NUMBER</td>
<td>I/O^4</td>
</tr>
<tr>
<td>OCCI BDouble</td>
<td>I/O</td>
</tr>
<tr>
<td>OCCI BFloat</td>
<td>I/O</td>
</tr>
<tr>
<td>OCCI Bytes</td>
<td>I/O^6</td>
</tr>
</tbody>
</table>
Data Conversions for LOB Data Types

### Table 5–6 Data Conversions for LOBs

<table>
<thead>
<tr>
<th>EXTERNAL DATATYPES</th>
<th>CLOB</th>
<th>BLOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR</td>
<td>I/O</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>I/O</td>
<td>-</td>
</tr>
<tr>
<td>LONG</td>
<td>I/O</td>
<td>-</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>I/O</td>
<td>-</td>
</tr>
<tr>
<td>STL String</td>
<td>I/O</td>
<td>-</td>
</tr>
<tr>
<td>RAW</td>
<td>-</td>
<td>I/O</td>
</tr>
<tr>
<td>VARRAW</td>
<td>-</td>
<td>I/O</td>
</tr>
<tr>
<td>OCCI Bytes</td>
<td>-</td>
<td>I/O</td>
</tr>
</tbody>
</table>

### Data Conversions for Date, Timestamp, and Interval Data Types

You can also use a character data type for the host variable used in a fetch or insert operation from or to a datetime or interval column. Oracle performs the conversion between the character data type and datetime/interval data type for you.

**See Also:** Oracle Database SecureFiles and Large Objects Developer’s Guide for an introduction to LOB data types.
These considerations apply when converting between Date, Timestamp, and Interval data types:

- When assigning a source with time zone to a target without a time zone, the time zone portion of the source is ignored. On assigning a source without a time zone to a target with a time zone, the time zone of the target is set to the session's default time zone.

- When assigning an Oracle DATE to a TIMESTAMP, the TIME portion of the DATE is copied over to the TIMESTAMP. When assigning a TIMESTAMP to Oracle DATE, the TIME portion of the result DATE is set to zero. This is done to encourage migration of Oracle DATE to ANSI compliant DATETIME data types.

- (When assigning an ANSI DATE to an Oracle DATE or a TIMESTAMP, the TIME portion of the Oracle DATE and the TIMESTAMP are set to zero. When assigning an Oracle DATE or a TIMESTAMP to an ANSI DATE, the TIME portion is ignored.

- When assigning a DATETIME to a character string, the DATETIME is converted using the session's default DATETIME format. When assigning a character string to a DATETIME, the string must contain a valid DATETIME value based on the session's default DATETIME format.

- When assigning a character string to an INTERVAL, the character string must be a valid INTERVAL character format.

- When converting from TSLTZ to CHAR, DATE, TIME, TIMESTAMP, and TSTZ, the value is adjusted to the session time zone.

- When converting from CHAR, DATE, and TIMESTAMP to TSLTZ, the session time zone is stored in memory.
When assigning TSLTZ to ANSI DATE, the time portion is 0.

When converting from TSTZ, the time zone that the time stamp is in is stored in memory.

When assigning a character string to an interval, the character string must be a valid interval character format.
This chapter describes how to retrieve metadata about result sets or the database as a whole.

This chapter contains these topics:

- **Overview of Metadata**
- **Describing Database Metadata**

### Overview of Metadata

Database objects have various attributes that describe them; you can obtain information about a particular schema object by performing a DESCRIBE operation. The result can be accessed as an object of the `Metadata` class by passing object attributes as arguments to the various methods of the `Metadata` class.

You can perform an explicit DESCRIBE operation on the database as a whole, on the types and properties of the columns contained in a `ResultSet` class, or on any of the following schema and subschema objects, such as tables, types, sequences, views, type attributes, columns, procedures, type methods, arguments, functions, collections, results, packages, synonyms, and lists.

You must specify the type of the attribute you are looking for. By using the `getAttributeCount()`, `getAttributeId()`, and `getAttributeType()` methods of the `MetaData` class, you can scan through each available attribute.

All DESCRIBE information is cached until the last reference to it is deleted. Users are in this way prevented from accidentally trying to access DESCRIBE information that is freed.

You obtain metadata by calling the `getMetaData()` method on the `Connection` class in case of an explicit describe, or by calling the `getColumnListMetaData()` method on the `ResultSet` class to get the metadata of the result set columns. Both methods return a `MetaData` object with the describing information. The `MetaData` class provides the `getxxx()` methods to access this information.

**See Also:** [Table 13–27, "Enumerated Values Used by MetaData Class" on page 13-128](#)

When performing DESCRIBE operations, be aware of the following issues:

- The `ATTR_TYPECODE` returns type codes that represent the type supplied when you created a new type by using the `CREATE TYPE` statement. These type codes are of the enumerated type `TypeCode`, which are represented by
Describing Database Metadata

Describing database metadata is equivalent to an explicit DESCRIBE operation. The object to describe must be an object in the schema. In describing a type, you call the getMetaData() method from the connection, passing the name of the object or a RefAny object. You must first initialize the environment in the OBJECT mode. The getMetaData() method returns an object of type MetaData. Each type of MetaData object has a list of attributes that are part of the describe tree. The describe tree can then be traversed recursively to point to subtrees that contain more information. More information about an object can be obtained by calling the getxxx() methods.

If you must construct a browser that describes the database and its objects recursively, then you can access information regarding the number of attributes for each object in the database (including the database), the attribute ID listing, and the attribute types listing. By using this information, you can recursively traverse the describe tree from the top node (the database) to the columns in the tables, the attributes of a type, the parameters of a procedure or function, and so on.

For example, consider the typical case of describing a table and its contents. You call the getMetaData() method from the connection, passing the name of the table to be described. The MetaData object returned contains the table information. Because you are aware of the type of the object you want to describe (table, column, type, collection, function, procedure, and so on), you can obtain the attribute list. You can retrieve the value into a variable of the type specified in the table by calling the corresponding getxxx() method.

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter Attributes on page 6-6</td>
<td>Attributes belonging to all elements</td>
</tr>
<tr>
<td>Table and View Attributes on page 6-6</td>
<td>Attributes belonging to tables and views</td>
</tr>
<tr>
<td>Procedure, Function, and Subprogram Attributes on page 6-7</td>
<td>Attributes belonging to procedures, functions, and package subprograms</td>
</tr>
<tr>
<td>Package Attributes on page 6-7</td>
<td>Attributes belonging to packages</td>
</tr>
<tr>
<td>Type Attributes on page 6-8</td>
<td>Attributes belonging to types</td>
</tr>
<tr>
<td>Type Attribute Attributes on page 6-9</td>
<td>Attributes belonging to type attributes</td>
</tr>
<tr>
<td>Type Method Attributes on page 6-9</td>
<td>Attributes belonging to type methods</td>
</tr>
<tr>
<td>Collection Attributes on page 6-10</td>
<td>Attributes belonging to collection types</td>
</tr>
<tr>
<td>Synonym Attributes on page 6-11</td>
<td>Attributes belonging to synonyms</td>
</tr>
<tr>
<td>Sequence Attributes on page 6-11</td>
<td>Attributes belonging to sequences</td>
</tr>
<tr>
<td>Column Attributes on page 6-12</td>
<td>Attributes belonging to columns of tables or views</td>
</tr>
<tr>
<td>Argument and Result Attributes on page 6-12</td>
<td>Attributes belonging to arguments / results</td>
</tr>
<tr>
<td>List Attributes on page 6-13</td>
<td>Attributes that designate the list type</td>
</tr>
</tbody>
</table>
Metadata Code Examples

This section provides code examples for using metadata:

- **Example 6–1, “How to Obtain Metadata About Attributes of a Simple Database Table”** on page 6-3
- **Example 6–2, “How to Obtain Metadata from a Column Containing User-Defined Types”** on page 6-4
- **Example 6–3, “How to Obtain Object Metadata from a Reference”** on page 6-5
- **Example 6–4, “How to Obtain Metadata About a Select List from a ResultSet Object”** on page 6-5

**Example 6–1 How to Obtain Metadata About Attributes of a Simple Database Table**

This example demonstrates how to obtain metadata about attributes of a simple database table:

```c++
/* Create an environment and a connection to the HR database */

/* Call the getMetaData method on the Connection object obtained */
MetaData emptab_metaData = connection->getMetaData("EMPLOYEES", MetaData::PTYPE_TABLE);
/* Now that you have the metadata information on the EMPLOYEES table, call the getxxx methods using the appropriate attributes */

/* Call getString */
cout<<"Schema:"
    <<emptab_metaData.getString(MetaData::ATTR_OBJ_SCHEMA)<<endl;

if(emptab_metaData.getInt(  
    emptab_metaData::ATTR_PTYPE)==MetaData::PTYPE_TABLE)  
    cout<<"EMPLOYEES is a table"<<endl;  
else  
    cout<<"EMPLOYEES is not a table"<<endl;

/* Call getInt to get the number of columns in the table */
int columnCount=emptab_metaData.getInt(MetaData::ATTR_NUM_COLS);
cout<<"Number of Columns:"<<columnCount<<endl;

/* Call getTimestamp to get the timestamp of the table object */
Timestamp tstamp = emptab_metaData.getTimestamp(MetaData::ATTR_TIMESTAMP);
/* Now that you have the value of the attribute as a Timestamp object, you can call methods to obtain the components of the timestamp */
int year;
unsigned int month, day;
tstamp.getData(year, month, day);

/* Call getVector for attributes of list type, such as ATTR_LIST_COLUMNS */
vector<MetaData>listOfColumns;
listOfColumns=emptab_metaData.getVector(MetaData::ATTR_LIST_COLUMNS);
```

**Table 6–1 (Cont.) Attribute Groupings**

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema Attributes</td>
<td>Attributes specific to schemas</td>
</tr>
<tr>
<td>Database Attributes</td>
<td>Attributes specific to databases</td>
</tr>
</tbody>
</table>
Each of the list elements represents a column metadata, so now you can access the column attributes:

```c++
for (int i=0; i<listOfColumns.size(); i++) {
    MetaData columnObj = listOfColumns[i];
    cout << "Column Name:" << columnObj.getString(MetaData::ATTR_NAME) << endl;
    cout << "Data Type:" << columnObj.getInt(MetaData::ATTR_DATA_TYPE) << endl;
    // and so on to obtain metadata on other column specific attributes
}
```

**Example 6-2  How to Obtain Metadata from a Column Containing User-Defined Types**

This example demonstrates how to obtain metadata from a column that contains user-defined types database table.

```c++
/* Create an environment and a connection to the HR database */
...
/* Call the getMetaData method on the Connection object obtained */
MetaData custtab_metaData = connection->getMetaData("CUSTOMERS", MetaData::PTYPE_TABLE);

/* Have metadata information on CUSTOMERS table; call the getxxx methods */
/* Call getString */
cout << "Schema:" << custtab_metaData.getString(MetaData::ATTR_OBJ_SCHEMA) << endl;
if (custtab_metaData.getInt(custtab_metaData::ATTR_PTYPE) == MetaData::PTYPE_TABLE)
    cout << "CUSTOMERS is a table" << endl;
else
    cout << "CUSTOMERS is not a table" << endl;

/* Call getVector to obtain list of columns in the CUSTOMERS table */
vector<MetaData> listOfColumns;
listOfColumns = custtab_metaData.getVector(MetaData::ATTR_LIST_COLUMNS);

/* Assuming metadata for column cust_address_typ is fourth element in list */
MetaData customer_address = listOfColumns[3];

/* Obtain the metadata for the customer_address attribute */
int typcode = customer_address.getInt(MetaData::ATTR_TYPECODE);
if (typcode == OCCI_TYPECODE_OBJECT)
    cout << "customer_address is an object type" << endl;
else
    cout << "customer_address is not an object type" << endl;

string objectName = customer_address.getString(MetaData::ATTR_OBJ_NAME);

/* Now that you have the name of the address object, the metadata of the attributes of the type can be obtained by using getMetaData on the connection by passing the object name */
MetaData address = connection->getMetaData(objectName);

/* Call getVector to obtain the list of the address object attributes */
vector<MetaData> attributeList = address.getVector(MetaData::ATT_LIST_TYPE_ATTRS);

/* and so on to obtain metadata on other address object specific attributes */
```
**Example 6–3  How to Obtain Object Metadata from a Reference**

This example demonstrates how to obtain metadata about an object when using a reference to it:

Type `ADDRESS(street VARCHAR2(50), city VARCHAR2(20));`
Table `Person(id NUMBER, addr REF ADDRESS);`

/* Create an environment and a connection to the HR database */
/* Call the getMetaData method on the Connection object obtained */
`MetaData perstab_metaData = connection->getMetaData(
    "Person", MetaData::PTYPE_TABLE);`

/* Now that you have the metadata information on the Person table, call the getxxx methods using the appropriate attributes */
/* Call getString */
`cout<<"Schema:"<<(perstab_metaData.getString(MetaData::ATTR_OBJ_SCHEMA))<<endl;`
if(perstab_metaData.getInt(perstab_metaData::ATTR_PTYPE)==MetaData::PTYPE_TABLE)
    cout<<"Person is a table"<<endl;
else
    cout<<"Person is not a table"<<endl;

/* Call getVector to obtain the list of columns in the Person table*/
`vector<MetaData>listOfColumns;`
`listOfColumns=perstab_metaData.getVector(MetaData::ATTR_LIST_COLUMNS);`

/* Each of the list elements represents a column metadata, so now get the data type of the column by passing ATTR_DATA_TYPE to getInt */
for(int i=0;i<numCols;i++)
{
    int dataType=colList[i].getInt(MetaData::ATTR_DATA_TYPE);
    /* If the data type is a reference, get the Ref and obtain the metadata about the object by passing the Ref to getMetaData */
    if(dataType==SQLT_REF)
        RefAny refTdo=colList[i].getRef(MetaData::ATTR_REF_TDO);

    /* Now you can obtain the metadata about the object as shown */
    MetaData tdo_metaData=connection->getMetaData(refTdo);

    /* Now that you have the metadata about the TDO, you can obtain the metadata about the object */
}

**Example 6–4  How to Obtain Metadata About a Select List from a ResultSet Object**

This example demonstrates how to obtain metadata about a select list from a ResultSet.

/* Create an environment and a connection to the database */
.../* Create a statement and associate it with a select clause */
`string sqlStmt="SELECT * FROM EMPLOYEES";`
`Statement *stmt=conn->createStatement(sqlStmt);`

/* Execute the statement to obtain a ResultSet */
`ResultSet *rset=stmt->executeQuery();`

/* Obtain the metadata about the select list */
vector<Meta<
/* The metadata is a column list and each element is a column metaData */
int dataType=cmd[i].getInt(MetaData::ATTR_DATA_TYPE);
...

The getMetaData method is called for the ATTR_COLLECTION_ELEMENT attribute only.

**Attribute Reference**

This section describes the attributes belonging to schema and subschema objects.

**Parameter Attributes**

All elements have some attributes specific to that element and some generic attributes. Table 6–2 describes the attributes that belong to all elements:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_OBJ_ID</td>
<td>Object or schema ID</td>
<td>unsigned int</td>
</tr>
<tr>
<td>ATTR_OBJ_NAME</td>
<td>Object, schema, or database name</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_OBJ_SCHEMA</td>
<td>Schema where object is located</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_OBJ_PTYPE</td>
<td>Type of information described by the parameter. Possible values are:</td>
<td>int</td>
</tr>
<tr>
<td>PTYPE_TABLE</td>
<td>Table</td>
<td></td>
</tr>
<tr>
<td>PTYPE_VIEW</td>
<td>View</td>
<td></td>
</tr>
<tr>
<td>PTYPE_PROC</td>
<td>Procedure</td>
<td></td>
</tr>
<tr>
<td>PTYPE_FUNC</td>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>PTYPE_PKG</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>PTYPE_TYPE</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>PTYPE_TYPE_ATTR</td>
<td>Attribute of a type</td>
<td></td>
</tr>
<tr>
<td>PTYPE_TYPE_COLL</td>
<td>Collection type information</td>
<td></td>
</tr>
<tr>
<td>PTYPE_TYPE_METHOD</td>
<td>A method of a type</td>
<td></td>
</tr>
<tr>
<td>PTYPE_SYN</td>
<td>Synonym</td>
<td></td>
</tr>
<tr>
<td>PTYPE_SEQ</td>
<td>Sequence</td>
<td></td>
</tr>
<tr>
<td>PTYPE_COL</td>
<td>Column of a table or view</td>
<td></td>
</tr>
<tr>
<td>PTYPE_ARG</td>
<td>Argument of a function or procedure</td>
<td></td>
</tr>
<tr>
<td>PTYPE_TYPE_ARG</td>
<td>Argument of a type method</td>
<td></td>
</tr>
<tr>
<td>PTYPE_TYPE_RESULT</td>
<td>Results of a method</td>
<td></td>
</tr>
<tr>
<td>PTYPE_SCHEMA</td>
<td>Schema</td>
<td></td>
</tr>
<tr>
<td>PTYPE_DATABASE</td>
<td>Database</td>
<td></td>
</tr>
<tr>
<td>ATTR_TIMESTAMP</td>
<td>The TIMESTAMP of the object this description is based on (Oracle DATE format).</td>
<td>Timestamp</td>
</tr>
</tbody>
</table>

The sections that follow list attributes specific to different types of elements.

**Table and View Attributes**

A parameter for a table or view (type PTYPE_TABLE or PTYPE_VIEW) has the following type-specific attributes described in Table 6–3:
The additional attributes belonging to tables are described in Table 6–4.

### Table 6–4 Attributes Specific to Tables

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_DBID</td>
<td>Data block address of the segment header</td>
<td>unsigned int</td>
</tr>
<tr>
<td>ATTR_TABLESPACE</td>
<td>Tablespace the table resides on</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_CLUSTERED</td>
<td>Identifies whether the table is clustered</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_PARTITIONED</td>
<td>Identifies whether the table is partitioned</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_INDEX_ONLY</td>
<td>Identifies whether the table is index only</td>
<td>bool</td>
</tr>
</tbody>
</table>

**Procedure, Function, and Subprogram Attributes**

A parameter for a procedure or function (type PTYPE_PROC or PTYPE_FUNC) has the type-specific attributes described in Table 6–5.

### Table 6–5 Attributes that Belong to Procedures or Functions

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_LIST_ARGUMENTS</td>
<td>Argument list; refer to <a href="#">List Attributes</a> on page 6-13.</td>
<td>vector&lt;MetaData&gt;</td>
</tr>
<tr>
<td>ATTR_IS_INVOKER_RIGHTS</td>
<td>Identifies whether the procedure or function has invoker’s rights.</td>
<td>int</td>
</tr>
</tbody>
</table>

The additional attributes belonging to package subprograms are described in Table 6–6.

### Table 6–6 Attributes that Belong to Package Subprograms

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_NAME</td>
<td>Name of procedure or function</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_OVERLOAD_ID</td>
<td>Overloading ID number (relevant in case the procedure or function is part of a package and is overloaded). Values returned may be different from direct query of a PL/SQL function or procedure.</td>
<td>int</td>
</tr>
</tbody>
</table>

**Package Attributes**

A parameter for a package (type PTYPE_PKG) has the type-specific attributes described in Table 6–7.
Type Attributes

A parameter for a type (type PTYPE_TYPE) has attributes described in Table 6–8.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_REF_TDO</td>
<td>Returns the in-memory ref of the type descriptor object for the type, if the column type is an object type.</td>
<td>RefAny</td>
</tr>
<tr>
<td>ATTR_TYPECODE</td>
<td>Type code. Can be:</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>■ OCCI_TYPECODE_OBJECT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ OCCI_TYPECODE_NAMEDCOLLECTION</td>
<td></td>
</tr>
<tr>
<td>ATTR_COLLECTION_TYPECODE</td>
<td>Type code of collection if type is collection; invalid otherwise. Can be:</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>■ OCCI_TYPECODE_VARRAY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ OCCI_TYPECODE_TABLE</td>
<td></td>
</tr>
<tr>
<td>ATTR_VERSION</td>
<td>A NULL-terminated string containing the user-assigned version</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_IS_FINAL_TYPE</td>
<td>Identifies whether this is a final type</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_INSTANTIABLE_TYPE</td>
<td>Identifies whether this is an instantiable type</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_SUBTYPE</td>
<td>Identifies whether this is a subtype</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_SUPERTYPE_SCHEMA_NAME</td>
<td>Name of the schema containing the supertype</td>
<td>string</td>
</tr>
<tr>
<td>ATTRSUPERTYPE_NAME</td>
<td>Name of the supertype</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_IS_INVOKER_RIGHTS</td>
<td>Identifies whether this type is invoker's rights</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_INCOMPLETE_TYPE</td>
<td>Identifies whether this type is incomplete</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_SYSTEM_TYPE</td>
<td>Identifies whether this is a system type</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_PREDEFINED_TYPE</td>
<td>Identifies whether this is a predefined type</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_TRANSIENT_TYPE</td>
<td>Identifies whether this is a transient type</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_SYSTEM_GENERATED_TYPE</td>
<td>Identifies whether this is a system-generated type</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_HAS_NESTED_TABLE</td>
<td>Identifies whether this type contains a nested table attribute</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_HASLOB</td>
<td>Identifies whether this type contains a LOB attribute</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_HASFILE</td>
<td>Identifies whether this type contains a FILE attribute</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_COLLECTION_ELEMENT</td>
<td>Handle to collection element</td>
<td>MetaData</td>
</tr>
<tr>
<td></td>
<td>Refer to Collection Attributes on page 6-10</td>
<td></td>
</tr>
<tr>
<td>ATTR_NUM_TYPE_ATTRS</td>
<td>Number of type attributes</td>
<td>unsigned int</td>
</tr>
<tr>
<td>ATTR_LIST_TYPE_ATTRS</td>
<td>List of type attributes</td>
<td>vector&lt;MetaData&gt;</td>
</tr>
<tr>
<td></td>
<td>Refer to List Attributes on page 6-13</td>
<td></td>
</tr>
</tbody>
</table>
Type Attribute Attributes

A parameter for an attribute of a type (type PTYPE_TYPE_ATTR) has the attributes described in Table 6–9.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_DATA_SIZE</td>
<td>Maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_TYPECODE</td>
<td>Type code</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_DATA_TYPE</td>
<td>Data type of the type attribute</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_NAME</td>
<td>A pointer to a string that is the type attribute name</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_PRECISION</td>
<td>Precision of numeric type attributes. If the precision is nonzero and scale is −127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented simply by NUMBER.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_SCALE</td>
<td>Scale of numeric type attributes. If the precision is nonzero and scale is −127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented simply as NUMBER.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_TYPE_NAME</td>
<td>A string that is the type name. The returned value contains the type name if the data type is SQLT_NTY or SQLT_REF. If the data type is SQLT_NTY, then the name of the named data type's type is returned. If the data type is SQLT_REF, then the type name of the named data type pointed to by the REF is returned.</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_SCHEMA_NAME</td>
<td>String with the schema name under which the type has been created</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_REF_TDO</td>
<td>Returns the in-memory REF of the TDO for the type, if the column type is an object type.</td>
<td>RefAny</td>
</tr>
<tr>
<td>ATTR_CHARSET_ID</td>
<td>Character set ID, if the type attribute is of a string or character type</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_CHARSET_FORM</td>
<td>Character set form, if the type attribute is of a string or character type</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_FSPRECISION</td>
<td>The fractional seconds precision of a datetime or interval</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_LFPRECISION</td>
<td>The leading field precision of an interval</td>
<td>int</td>
</tr>
</tbody>
</table>

Type Method Attributes

A parameter for a method of a type (type PTYPE_TYPE_METHOD) has the attributes described in Table 6–10.
### Table 6–10 Attributes that Belong to Type Methods

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_NAME</td>
<td>Name of method (procedure or function)</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_ENCAPSULATION</td>
<td>Encapsulation level of the method; can be:</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>- OCCI_TYPEENCAP_PRIVATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- OCCI_TYPEENCAP_PUBLIC</td>
<td></td>
</tr>
<tr>
<td>ATTR_LIST_ARGUMENTS</td>
<td>Argument list</td>
<td>vector&lt;MetaData&gt;</td>
</tr>
<tr>
<td>ATTR_IS_CONSTRUCTOR</td>
<td>Identifies whether the method is a constructor</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_DESTRUCTOR</td>
<td>Identifies whether the method is a destructor</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_OPERATOR</td>
<td>Identifies whether the method is an operator</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_SELFISH</td>
<td>Identifies whether the method is selfish</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_MAP</td>
<td>Identifies whether the method is a map method</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_ORDER</td>
<td>Identifies whether the method is an order method</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_RNDS</td>
<td>Identifies whether &quot;Read No Data State&quot; is set for</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td>the method</td>
<td></td>
</tr>
<tr>
<td>ATTR_IS_RNPS</td>
<td>Identifies whether &quot;Read No Process State&quot; is set</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td>for the method</td>
<td></td>
</tr>
<tr>
<td>ATTR_IS_WNDS</td>
<td>Identifies whether &quot;Write No Data State&quot; is set</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td>for the method</td>
<td></td>
</tr>
<tr>
<td>ATTR_IS_WNPS</td>
<td>Identifies whether &quot;Write No Process State&quot; is set</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td>for the method</td>
<td></td>
</tr>
<tr>
<td>ATTR_IS_FINAL_METHOD</td>
<td>Identifies whether this is a final method</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_INSTANTIABLE_METHOD</td>
<td>Identifies whether this is an instantiable method</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_IS_OVERRIDING_METHOD</td>
<td>Identifies whether this is an overriding method</td>
<td>bool</td>
</tr>
</tbody>
</table>

### Collection Attributes

A parameter for a collection type (type PTYPE_COLL) has the attributes described in Table 6–11.

### Table 6–11 Attributes that Belong to Collection Types

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_DATA_SIZE</td>
<td>Maximum size of the type attribute. This length is returned in bytes and not characters for</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>strings and raws. Returns 22 for NUMBER.</td>
<td></td>
</tr>
<tr>
<td>ATTR_TYPECODE</td>
<td>Typecode.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_DATA_TYPE</td>
<td>The data type of the type attribute.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_NUM_ELEMENTS</td>
<td>Number of elements in an array; only valid for collections that are arrays.</td>
<td>unsigned int</td>
</tr>
<tr>
<td>ATTR_NAME</td>
<td>A pointer to a string that is the type attribute name.</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_PRECISION</td>
<td>Precision of numeric type attributes. If the precision is nonzero and scale is -127, then it</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented</td>
<td></td>
</tr>
<tr>
<td></td>
<td>simply as NUMBER.</td>
<td></td>
</tr>
</tbody>
</table>
Synonym Attributes

A parameter for a synonym (type PTYPE_SYN) has the attributes described in Table 6–12.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_SCALE</td>
<td>Scale of numeric type attributes. If the precision is nonzero and scale is -127, then it is a FLOAT; otherwise a NUMBER (p, s). If precision is 0, then NUMBER (p, s) can be represented simply as NUMBER.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_TYPE_NAME</td>
<td>String that is the type name. The returned value contains the type name if the data type is SQLT_NTY or SQLT_REF. If the data type is SQLT_NTY, then the name of the named data type's type is returned. If the data type is SQLT_REF, then the type name of the named data type pointed to by the REF is returned</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_SCHEMA_NAME</td>
<td>String with the schema name under which the type has been created</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_REF_TDO</td>
<td>Returns the in memory REF of the TDO for the type.</td>
<td>RefAny</td>
</tr>
<tr>
<td>ATTR_CHARSET_ID</td>
<td>Typecode.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_CHARSET_FORM</td>
<td>The data type of the type attribute.</td>
<td>int</td>
</tr>
</tbody>
</table>

Sequence Attributes

A parameter for a sequence (type PTYPE_SEQ) has the attributes described in Table 6–13.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_OBJID</td>
<td>Object ID</td>
<td>unsigned int</td>
</tr>
<tr>
<td>ATTR_SCHEMA_NAME</td>
<td>Null-terminated string containing the schema name of the synonym translation</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_NAME</td>
<td>Null-terminated string containing the object name of the synonym translation</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_LINK</td>
<td>Null-terminated string containing the database link name of the synonym translation</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_MIN</td>
<td>Minimum value (in Oracle number format)</td>
<td>Number</td>
</tr>
<tr>
<td>ATTR_MAX</td>
<td>Maximum value (in Oracle number format)</td>
<td>Number</td>
</tr>
<tr>
<td>ATTR_INCR</td>
<td>Increment (in Oracle number format)</td>
<td>Number</td>
</tr>
<tr>
<td>ATTR_CACHE</td>
<td>Number of sequence numbers cached; zero if the sequence is not a cached sequence (in Oracle number format)</td>
<td>Number</td>
</tr>
<tr>
<td>ATTR_ORDER</td>
<td>Identifies whether the sequence is ordered</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_HW_MARK</td>
<td>High-water mark (in Oracle number format)</td>
<td>Number</td>
</tr>
</tbody>
</table>
Column Attributes

A parameter for a column of a table or view (type PTYPE_COL) has the attributes described in Table 6–14.

Table 6–14  Attributes that Belong to Columns of Tables or Views

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_DATA_SIZE</td>
<td>Column length in codepoints. The number of codepoints allowed in the column.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_DATA_TYPE</td>
<td>Type of length semantics of the column. Valid values are 0 for byte-length semantics and 1 for codepoint-length semantics.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_NAME</td>
<td>Maximum size of the column. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_PRECISION</td>
<td>The data type of the column.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_SCALE</td>
<td>Pointer to a string that is the column name</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_IS_NULL</td>
<td>Returns TRUE if the attribute is NULL; FALSE otherwise.</td>
<td>bool</td>
</tr>
<tr>
<td>ATTR_TYPE_NAME</td>
<td>Scale of numeric columns. If the precision is nonzero and scale is −127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented simply as NUMBER.</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_SCHEMA_NAME</td>
<td>Returns 0 if null values are not permitted for the column</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_REF_TDO</td>
<td>Returns a string that is the type name. The returned value contains the type name if the data type is SQLT_NTY or SQLT_REF. If the data type is SQLT_NTY, then the name of the named data type's type is returned. If the data type is SQLT_REF, then the type name of the named data type pointed to by the REF is returned.</td>
<td>RefAny</td>
</tr>
<tr>
<td>ATTR_CHARSET_ID</td>
<td>Returns a string with the schema name under which the type has been created.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_CHARSET_FORM</td>
<td>The REF of the TDO for the type, if the column type is an object type</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_DATA_SIZE</td>
<td>Size of the data type of the argument. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.</td>
<td>int</td>
</tr>
</tbody>
</table>

Argument and Result Attributes

A parameter for an argument or a procedure or function type (type PTYPE_ARG), for a type method argument (type PTYPE_TYPE_ARG), or for method results (type PTYPE_TYPE_RESULT) has the attributes described in Table 6–15.

Table 6–15  Attributes that Belong to Arguments / Results

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_NAME</td>
<td>Returns a pointer to a string which is the argument name</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_POSITION</td>
<td>Position of the argument in the argument list. Always returns 0.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_TYPECODE</td>
<td>Typecode.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_DATA_TYPE</td>
<td>Data type of the argument.</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_DATA_SIZE</td>
<td>Size of the data type of the argument. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.</td>
<td>int</td>
</tr>
</tbody>
</table>
List Attributes

A list type of attribute can be described for all the elements in the list. In case of a function argument list, position 0 has a parameter for return values (PTYPE_ARG).

The list is described iteratively for all the elements. The results are stored in a C++ vector<MetaData>. Call the getVector() method to describe list type of attributes. Table 6–16 displays the list attributes.
Table 6–16  Values for ATTR_LIST_TYPE

<table>
<thead>
<tr>
<th>Possible Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_LIST_COLUMNS</td>
<td>Column list</td>
</tr>
<tr>
<td>ATTR_LIST_ARGUMENTS</td>
<td>Procedure or function arguments list</td>
</tr>
<tr>
<td>ATTR_LIST_SUBPROGRAMS</td>
<td>Subprogram list</td>
</tr>
<tr>
<td>ATTR_LIST_TYPE_ATTRIBUTES</td>
<td>Type attribute list</td>
</tr>
<tr>
<td>ATTR_LIST_TYPE_METHODS</td>
<td>Type method list</td>
</tr>
<tr>
<td>ATTR_LIST_OBJECTS</td>
<td>Object list within a schema</td>
</tr>
<tr>
<td>ATTR_LIST_SCHEMAS</td>
<td>Schema list within a database</td>
</tr>
</tbody>
</table>

Schema Attributes

A parameter for a schema type (type PTYPE_SCHEMA) has the attributes described in Table 6–17.

Table 6–17  Attributes Specific to Schemas

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_LIST_OBJECTS</td>
<td>List of objects in the schema</td>
<td>string</td>
</tr>
</tbody>
</table>

Database Attributes

A parameter for a database (type PTYPE_DATABASE) has the attributes described in Table 6–18.

Table 6–18  Attributes Specific to Databases

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_VERSION</td>
<td>Database version</td>
<td>string</td>
</tr>
<tr>
<td>ATTR_CHARSET_ID</td>
<td>Database character set ID from the server handle</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_NCHARSET_ID</td>
<td>Database native character set ID from the server handle</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_LIST_SCHEMAS</td>
<td>List of schemas (type PTYPE_SCHEMA) in the database</td>
<td>vector&lt;MetaData&gt;</td>
</tr>
<tr>
<td>ATTR_MAX_PROC_LEN</td>
<td>Maximum length of a procedure name</td>
<td>unsigned int</td>
</tr>
<tr>
<td>ATTR_MAX_COLUMN_LEN</td>
<td>Maximum length of a column name</td>
<td>unsigned int</td>
</tr>
<tr>
<td>ATTR_CURSOR_COMMIT_BEHAVIOR</td>
<td>How a COMMIT operation affects cursors and prepared statements in the database; values are:</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>■ OCCI_CURSOR_OPEN for preserving cursor state as before the commit operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ OCCI_CURSOR_CLOSED for cursors that are closed on COMMIT, although the application can execute the statement for the second time without preparing it again</td>
<td></td>
</tr>
<tr>
<td>ATTR_MAX_CATALOG_NAMELEN</td>
<td>Maximum length of a catalog (database) name</td>
<td>int</td>
</tr>
<tr>
<td>ATTR_CATALOG_LOCATION</td>
<td>Position of the catalog in a qualified table; values are:</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>■ OCCI_CL_START</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ OCCI_CL_END</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
<td>Attribute Data Type</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>ATTR_SAVEPOINT_SUPPORT</td>
<td>Identifies whether the database supports savepoints; values are:</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>■ OCCI_SP_SUPPORTED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ OCCI_SP_UNSUPPORTED</td>
<td></td>
</tr>
<tr>
<td>ATTR_NOWAIT_SUPPORT</td>
<td>Identifies whether the database supports the nowait clause; values are:</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>■ OCCI_NW_SUPPORTED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ OCCI_NW_UNSUPPORTED</td>
<td></td>
</tr>
<tr>
<td>ATTR_AUTOCOMMIT DDL</td>
<td>Identifies whether the autocommit mode is required for DDL statements; values are:</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>■ OCCI_AC_DDL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ OCCI_NO_AC_DDL</td>
<td></td>
</tr>
<tr>
<td>ATTR_LOCKING_MODE</td>
<td>Locking mode for the database; values are:</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>■ OCCI_LOCK_IMMEDIATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ OCCI_LOCK_DELAYED</td>
<td></td>
</tr>
</tbody>
</table>
This chapter provides an overview of LOBs and their use in OCCI.

This chapter contains these topics:

- Overview of LOBs
- Creating LOBs in OCCI Applications
- Restricting the Opening and Closing of LOBs
- Reading and Writing LOBs
- Using Objects with LOB Attributes
- Using SecureFiles

See also: Oracle Database SecureFiles and Large Objects Developer’s Guide for extensive information about LOBs

## Overview of LOBs

Oracle C++ Call Interface includes classes and methods for performing operations on large objects, LOBs. LOBs are either internal or external depending on their location with respect to the database.

## Introducing Internal LOBs

Internal LOBs are stored inside database tablespaces in a way that optimizes space and enables efficient access. Internal LOBs use copy semantics and participate in the transactional model of the server. You can recover internal LOBs after transaction or media failure, and any changes to an internal LOB value can be committed or rolled back. There are three SQL data types for defining instances of internal LOBs:

- **BLOB**: A LOB whose value is composed of unstructured binary (raw) data
- **CLOB**: A LOB whose value is composed of character data that corresponds to the database character set defined for the Oracle database
- **NCLOB**: A LOB whose value is composed of character data that corresponds to the national character set defined for the Oracle database

The copy semantics for LOBs dictate that when you insert or update a LOB with a LOB from another row in the same table, both the LOB locator and the LOB value are copied. In other words, each row has a copy of the LOB value.
Introducing External LOBs

BFILEs are large binary (raw) data objects data stored in operating system files outside database tablespaces; therefore, they are referred to as external LOBs. These files use reference semantics, where only the locator for the LOB is reproduced when inserting or updating in the same table. Apart from conventional secondary storage devices such as hard disks, BFILEs may also be located on tertiary block storage devices such as CD-ROMs, PhotoCDs and DVDs. The BFILE data type allows read-only byte stream access to large files on the file system of the database server. Oracle can access BFILEs if the underlying server operating system supports stream mode access to these files.

External LOBs do not participate in transactions. Any support for integrity and durability must be provided by the underlying file and operating systems. An external LOB must reside on a single device; it may not be striped across a disk array.

Storing LOBs

The size of the LOB value, among other things, dictates where it is stored. The LOB value is either stored inline with the row data or outside the row.

- **Locator storage:** a LOB locator, a pointer to the actual location of the LOB value, is stored inline with the row data and indicates where the LOB value is stored.

  For internal LOBs, the LOB column stores a locator to the LOB value stored in a database tablespace. Each internal LOB column and attribute for a particular row has its own unique LOB locator and a distinct copy of the LOB value stored in the database tablespace.

  For external LOBs, the LOB column stores a locator to the external operating system file that houses the BFILE. Each external LOB column and attribute for a given row has its own BFILE locator. However, two different rows can contain a BFILE locator that points to the same operating system file.

- **Inline storage:** Data stored in a LOB is termed the LOB value. The value of an internal LOB may or may not be stored inline with the other row data. If you do not set DISABLE STORAGE IN ROW, and if the internal LOB value is less than approximately 4,000 bytes, then the value is stored inline. Otherwise, it is stored outside the row.

  Since LOBs are intended to be large objects, inline storage is only relevant if your application mixes small and large LOBs. The LOB value is automatically moved out of the row once it extends beyond approximately 4,000 bytes.

Creating LOBs in OCCI Applications

Follow these steps to use LOBs in your application:

- Initialize a new LOB locator in the database.

- Assign a value to the LOB. In case of BFILEs, assign a reference to a valid external file.

- To access and manipulate LOBs, see the OCCI classes that implement the methods for using LOBs in an application. All are detailed in Chapter 13, "OCCI Application Programming Interface":
  - Bfile Class on page 13-20 contains the APIs for BFILEs, as summarized in Table 13-7.
- **Blob Class** on page 13-26 contains the APIs for BLOBs, as summarized in Table 13–8.

- **Clob Class** on page 13-38 contains the APIs for CLOBs and NCLOBs, as summarized in Table 13–10.

Whenever you want to modify an internal LOB column or attribute using write, copy, trim, and similar operations, you must lock the row that contains the target LOB. Use a `SELECT...FOR UPDATE` statement to select the LOB locator.

A transaction must be open before a LOB write command succeeds. Therefore, you must write the data before committing a transaction (since `COMMIT` closes the transaction). Otherwise, you must lock the row again by reissuing the `SELECT...FOR UPDATE` statement. Each of the LOB class implementations in OCCI have `open()` and `close()` methods. To check whether a LOB is open, call the `isOpen()` method of the class.

The methods `open()`, `close()` and `isOpen()` should also be used to mark the beginning and end of a series of LOB operations. Whenever a LOB modification is made, it triggers updates on extensible indexes. If these modifications are made within `open()...close()` code blocks, the individual triggers are disabled until after the `close()` call, and then all are issued at the same time. This implementation enables the efficient processing of maintenance operations, such as updating indexes, when the LOBs are closed. However, this also means that extensible indexes are not valid during the execution of the `open()...close()` code block.

Note that for internal LOBs, the concept of openness is associated with the LOB and not the LOB locator. The LOB locator does not store any information about whether the LOB to which it refers is open. It is possible for multiple LOB locators to point to the same open LOB. However, for external LOBs, openness is associated with a specific external LOB locator. Therefore, multiple `open()` calls can be made on the same BFILE using different external LOB locators.

### Restricting the Opening and Closing of LOBs

The definition of a transaction within which an open LOB value must be closed is one of the following:

- Between `SET TRANSACTION` and `COMMIT`
- Between `DATA MODIFYING DML` and `COMMIT`
- Between `SELECT...FOR UPDATE` and `COMMIT`
- Within an autonomous transaction block

The LOB opening and closing mechanism has the following restrictions:

- An application must close all previously opened LOBs before committing a transaction. Failing to do so results in an error. If a transaction is rolled back, then all open LOBs are discarded along with the changes made, so associated triggers are not fired.

- While there is no limit to the number of open internal LOBs, there is a limit on the number of open files. Note that assigning an opened locator to another locator does not count as opening a new LOB.

- It is an error to open or close the same internal LOB twice within the same transaction, either with different locators or with the same locator.

- It is an error to close a LOB that has not been opened.
Reading and Writing LOBs

There are two general methods for reading and writing LOBs: non-streamed, and streamed.

Reading LOBs

Example 7–1 illustrates how to get data from a non-NULL internal LOB, using a non-streamed method. This method requires that you keep track of the read offset and the amount remaining to be read, and pass these values to the read() method.

Example 7–1  How to Read Non-Streamed BLOBs

```c
ResultSet *rset=stmt->executeQuery("SELECT ad_composite FROM print_media WHERE product_id=6666");
while(rset->next())
{
    Blob blob=rset->getBlob(1);
    if(blob.isNull())
        cerr <<"Null Blob"<<endl;
    else
    {
        blob.open(OCCI_LOB_READONLY);
        const unsigned int BUFSIZE=100;
        char buffer[BUFSIZE];
        unsigned int readAmt=BUFSIZE;
        unsigned int offset=1;

        //reading readAmt bytes from offset 1
        blob.read(readAmt,buffer,BUFSIZE,offset);

        //process information in buffer
        ...  
        blob.close();
    }
}
stmt->closeResultSet(rset);
```

Example 7–2 is similar as it demonstrates how to read data from a BFILE, where the BFILE locator is not NULL, by using a non-streamed read.

Example 7–2  How to Read Non-Streamed BFILES

```c
ResultSet *rset=stmt->executeQuery("SELECT ad_graphic FROM print_media WHERE product_id=6666");
while(rset->next())
{
    Bfile file=rset->getBfile(1);
    if(file.isNull())
        cerr <<"Null Bfile"<<endl;
    else
    {
        //display the directory alias and the file name of the BFILE
        cout <<"File Name:"<<file.getFileName()<<endl;
        cout <<"Directory Alias:"<<file.getDirAlias()<<endl;

        if(file.fileExists())
        {
            unsigned int length=file.length();
```
Reading and Writing LOBs

In contrast to Example 7–1 and Example 7–2, the streamed reading demonstrated in Example 7–3 on a non-NULL BLOB does not require keeping track of the offset.

Example 7–3  How to Read Streamed BLOBs

```
char *buffer=new char[length];
bfile.read(length, buffer, length, 1);
//read all the contents of the BFILE into buffer, then process
...
delete[] buffer;
}
else
  cerr <<'File does not exist'<<endl;
}
}
stmt->closeResultSet(rset);
```

Writing LOBs

Example 7–4 demonstrates how to write data to an internal non-NULL LOB by using a non-streamed write. The writeChunk() method is enclosed by the open() and close() methods; it operates on a LOB that is currently open and ensures that triggers do not fire for every chunk read. The write() method can be used for the writeChunk() method; however, the write() method implicitly opens and closes the LOB.

Example 7–4  How to Write Non-Streamed BLOBs

```
char *buffer=new char[length];
bfile.read(length, buffer, length, 1);
//read all the contents of the BFILE into buffer, then process
...
delete[] buffer;
}
else
  cerr <<'File does not exist'<<endl;
}
}
stmt->closeResultSet(rset);
```
Example 7–5 demonstrates how to write data to an internal LOB that is populated by using a streamed write.

**Example 7–5  How to Write Streamed BLOBs**

```cpp
ResultSet *rset=stmt->executeQuery("SELECT ad_composite FROM print_media
WHERE product_id=6666 FOR UPDATE");
while(rset->next())
{
    Blob blob=rset->getBlob(1);
    if(blob.isNull())
        cerr <<"Null Blob"<<endl;
    else
    {
        char buffer[BUFSIZE];
        Stream *outstream=blob.getStream(1,0);

        //writing from buffer beginning at offset 1 until
        //a writeLastBuffer() method is issued.
        //contents of buffer are replaced after each writeBuffer(),
        //typically with an fread()
        while(<fread 'BUFSIZE' bytes into buffer succeeds>)
        {
            outstream->writeBuffer(buffer,BUFSIZE);
            outstream->writeLastBuffer(buffer,<remaining amt>);
        }
        blob.closeStream(outstream);
    }
}
stmt->closeResultSet(rset);
conn->commit();
```

**Enhancing the Performance of LOB Reads and Writes**

Reading and writing of internal LOBs can be improved by using either `getChunkSize()` method.
Using the getChunkSize() Method

The getChunkSize() method returns the usable chunk size in bytes for BLOBs, and in characters for CLOBs and NCLOBs. Performance improves when a read or a write begins on a multiple of the usable chunk size, and the request size is also a multiple of the usable chunk size. You can specify the chunk size for a LOB column when you create a table that contains the LOB.

Calling the getChunkSize() method returns the usable chunk size of the LOB. An application can batch a series of write operations until an entire chunk can be written, rather than issuing multiple LOB write calls that operate on the same chunk.

To read through the end of a LOB, use the read() method with an amount of 4 GB. This avoids the round-trip involved with first calling the getLength() method because the read() method with an amount of 4 GB reads until the end of the LOB is reached.

For LOBs that store variable width characters, the GetChunkSize() method returns the number of Unicode characters that fit in a LOB chunk.

Updating LOBs

To update a value of a LOB in the database, you must assign the new value to the LOB, execute a SQL UPDATE command in the database, and then commit the transaction. Example 7–6 demonstrates how to update an existing CLOB (in this case, by setting it to empty), while Example 7–7 demonstrates how to update a BFILE.

Example 7–6   Updating a CLOB Value

Clob clob(conn);
clob.setEmpty();
stmt->setSQL("UPDATE print_media SET ad_composite = :1
             WHERE product_id=6666");
stmt->setClob(1, clob);
stmt->executeUpdate();
conn->commit();

Example 7–7   Updating a BFILE Value

Bfile bfile(conn);
bfile.setName("MEDIA_DIR", "img1.jpg");
stmt->setSQL("UPDATE print_media SET ad_graphic = :1
             WHERE product_id=6666");
stmt->setBfile(1, bfile);
stmt->executeUpdate();
conn->commit();

Reading and Writing Multiple LOBs

As of Oracle Database 10g Release 2, OCCI has new interfaces that enhance application performance while reading and writing multiple LOBs, such as Bfiles, Blobs, Clobs and NClobs.

These interfaces have several advantages over the standard methods for reading and writing a single LOB at a time:

- Reading and writing multiple LOBs through OCCI in a single server round-trip improves performance by decreasing I/O time between the application and the back end.
The new APIs provide support for LOBs that are larger than the previous limit of 4 GB. The new interfaces accept the oraub8 data type for amount, offsets, buffer and length parameters. These parameters are mapped to the appropriate 64-bit native data type, which is determined by the compiler and the operating system.

For Clob-related methods, the user can specify the data amount read or written in terms of character counts or byte counts.

New APIs for this features are described in Chapter 13, "OCCI Application Programming Interface", section on Connection Class, and include readVectorOfBfiles() on page 13-56, readVectorOfBlobs() on page 13-56, readVectorOfClob() on page 13-57 (overloaded to support general character sets, and the UTF16 character set in particular), writeVectorOfBlobs() on page 13-60, and writeVectorOfClob() on page 13-60 (overloaded to support general character sets, and the UTF16 character set in particular).

Using the Interfaces for Reading and Writing Multiple LOBs

Each of the readVectorOfxxx() and writeVectorOfxxx() interface uses the following parameters:

- conn, a Connection class object
- vec, a vector of LOB objects: Bfile, Blob, Clob, or NClob
- byteAmts, array of amounts, in bytes, for reading or writing
- charAmts, array of amounts, in characters, for reading or writing (only applicable for Clobs and NClobs)
- offsets, array of offsets, in bytes for Bfiles and Blobs, and in characters for Clobes and NCLobs
- buffers, array of buffer pointers
- bufferLengths, array of buffer lengths.

If there are errors in either reading or writing of one of the LOBs in the vector, the whole operation is cancelled. The byteAmts or charAmts parameters should be checked to determine the actual number of bytes or characters read or written.

Using Objects with LOB Attributes

An OCCI application can use the operator new() to create a persistent object with a LOB attribute. By default, all LOB attributes are constructed by using the default constructor, and are initialized to NULL.

Example 7–8 demonstrates how to create and use persistent objects with internal LOB attributes. Example 7–9 demonstrates how to create and use persistent objects with external LOB attributes.

Example 7–8  How to Use a Persistent Object with a BLOB Attribute

1. Create a persistent object with a BLOB attribute:

   ```
   Person *p=new(conn,"PERSON_TAB")Person();
   p->imgBlob = Blob(conn);
   ```

2. Either initialize the Blob object to empty:

   ```
   p->imgBlob.setEmpty();
   ```
   Or set it to some existing value
3. Mark the Blob object as dirty:
   
   ```
   p->markModified();
   ```

4. Flush the object:
   
   ```
   p->flush();
   ```

5. Repin the object after obtaining a REF to it, thereby retrieving a refreshed version of the object from the database and acquiring an initialized LOB:
   
   ```
   Ref<Person> r = p->getRef();
   delete p;
   p = r.ptr();
   ```

6. Write the data:
   
   ```
   p->imgBlob.write( ... );
   ```

---

**Example 7–9  How to Use a Persistent Object with a BFILE Attribute**

1. Create a persistent object with a BFILE attribute:
   
   ```
   Person *p=new(conn,"PERSON_TAB")Person();
   p->imgBFile = BFile(conn);
   ```

2. Initialize the Bfile object:
   
   ```
   p->setName(directory_alias, file_name);
   ```

3. Mark the Bfile object as dirty:
   
   ```
   p->markModified();
   ```

4. Flush the object:
   
   ```
   p->flush();
   ```

5. Read the data:
   
   ```
   p->imgBfile.read( ... );
   ```

---

**Using SecureFiles**

Introduced with Oracle Database 11g Release 1, SecureFiles LOBs add powerful new features for LOB compression, encryption, and deduplication.

**See Also:** *Oracle Database SecureFiles and Large Objects Developer’s Guide*

---

**Using SecureFile Compression**

SecureFiles compression enables server-side compression of LOB data, transparent to the application. Using SecureFiles compression saves storage space with minimal impact on reading and updating performance for SecureFiles LOB data.

---

**Using SecureFiles Encryption**

SecureFiles introduce a new encryption capability for LOB data and extend Transparent Data Encryption by enabling efficient random read and write access to encrypted SecureFiles LOBs.
Using SecureFiles Deduplication

SecureFiles deduplication allows the Oracle Database to automatically detect duplicate LOB data, and to conserve space by storing a single copy of the SecureFiles LOB.

Combining SecureFiles Compression, Encryption, and Deduplication

You can combine compression, encryption and deduplication in any combination. Oracle Database applies these features according to the following rules:

- Deduplicate detection, if enabled, is performed before compression and encryption. This prevents potentially unnecessary and expensive compression and encryption operations on duplicate SecureFiles LOBs.
- Compression is performed before encryption, to allow for the highest possible compression ratios.

SecureFiles LOB Types and Constants

The following types for SecureFiles LOBs enable additional flexibility for compression, encryption, and deduplication. Table 7–1 lists options for the LobOptionType, while Table 7–2 lists options for the LobOptionValue.

<table>
<thead>
<tr>
<th>Table 7–1</th>
<th>Values of Type LobOptionType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>OCCI_LOB_OPT_COMPRESS</td>
<td>Compression option type</td>
</tr>
<tr>
<td>OCCI_LOB_OPT_ENCRYPT</td>
<td>Encryption option type</td>
</tr>
<tr>
<td>OCCI_LOB_OPT_DEDUPLICATE</td>
<td>Deduplicate option type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 7–2</th>
<th>Values of Type LobOptionValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>OCCI_LOB_COMPRESS_OFF</td>
<td>Turns off SecureFiles compression</td>
</tr>
<tr>
<td>OCCI_LOB_COMPRESS_ON</td>
<td>Turns on SecureFiles compression</td>
</tr>
<tr>
<td>OCCI_LOB_ENCRYPT_OFF</td>
<td>Turns off SecureFiles encryption</td>
</tr>
<tr>
<td>OCCI_LOB_ENCRYPT_ON</td>
<td>Turns on SecureFiles encryption</td>
</tr>
<tr>
<td>OCCI_LOB_DEDUPLICATE_OFF</td>
<td>Turns off SecureFiles deduplication</td>
</tr>
<tr>
<td>OCCI_LOB_DEDUPLICATE_ON</td>
<td>Turns off LOB deduplication</td>
</tr>
</tbody>
</table>
This chapter discusses the Object Type Translator (OTT) utility, which is used to map database object types, LOB types, and named collection types to C++ class declarations for use in OCCI applications.

This chapter contains these topics:

- Overview of the Object Type Translator Utility
- Using the OTT Utility
- Creating Types in the Database
- Invoking the OTT Utility
- Using the INTYPE File
- OTT Utility Data Type Mappings
- Overview of the OUTTYPE File
- The OTT Utility and OCCI Applications
- Carrying Forward User Added Code

See Also: $ORACLE_HOME/rdbms/demo for a complete code listing of the demonstration program used in this chapter, and the class and method implementation generated by the OTT utility.

Overview of the Object Type Translator Utility

The Object Type Translator (OTT) utility assists in the development of applications that make use of user-defined types in an Oracle database server.

You can create object types using the SQL CREATE TYPE statement. The definitions of these types are stored in the database, and can be subsequently used to create database tables. Once these tables are populated, an OCCI programmer can access objects stored in the tables.

An application that accesses object data must be able to represent the data in a host language format. This is accomplished by representing object types classes in C++.

You could code structures or classes manually to represent database object types, but this is time-consuming and error-prone. The OTT utility simplifies this step by automatically generating the appropriate classes for C++.

For OCCI, the application must include and link the following files:

- Include the header file containing the generated class declarations
Using the OTT Utility

To translate database types to C++ representation, you must explicitly invoke the OTT utility. OCCI programmers must register the mappings with the environment. This function is generated by the OTT utility.

On most operating systems, the OTT utility is invoked on the command line. It takes as input an INTYPE file, and generates an OUTTYPE file, one or more C++ header files that contain the prototype information, and additional C++ method files that register generated mappings.

Example 8–1 How to Use the OTT Utility

The following command invokes the OTT utility and generates C++ classes. OTT attempts to connect with username demouser; the system prompts for the password.

```
ott userid=demouser intype=demoin.typ outtype=demoout.typ code=cpp
       hfile=demo.h cppfile=demo.cpp mapfile=RegisterMappings.cpp
```

OTT utility uses the demoin.typ file as the INTYPE file, and the demoout.typ file as the OUTTYPE file. The resulting declarations are output to the file demo.h in C++, specified by the CODE=cpp parameter, the method implementations written to the file demo.cpp, and the functions to register mappings is written to RegisterMappings.cpp with its prototype written to RegisterMappings.h.

See Also: Extending C++ Classes on page 8-26 for a complete C++ example

Creating Types in the Database

The first step in using the OTT utility is to create object types or named collection types and store them in the database. This is accomplished by the SQL CREATE TYPE statement.

Example 8–2 Object Creation Statements of the OTT Utility

```
CREATE TYPE FULL_NAME AS OBJECT (first_name CHAR(20), last_name CHAR(20));
CREATE TYPE ADDRESS AS OBJECT (state CHAR(20), zip CHAR(20));
CREATE TYPE ADDRESS_TAB AS VARRAY(3) OF REF ADDRESS;
CREATE TYPE PERSON AS OBJECT (id NUMBER, name FULL_NAME, curr_addr REF ADDRESS,
       prev_addr_1 ADDRESS_TAB) NOT FINAL;
CREATE TYPE STUDENT UNDER PERSON (school_name CHAR(20));
```

Invoking the OTT Utility

After creating types in the database, the next step is to invoke the OTT utility.
Specifying OTT Parameters

You can specify OTT parameters either on the command line or in a configuration file. Certain parameters can also be specified in the INTYPE file.

If you specify a parameter in multiple locations, then its value on the command line takes precedence over its value in the INTYPE file. The value in the INTYPE file takes precedence over its value in a user-defined configuration file, which takes precedence over its value in the default configuration file.

Parameter precedence then is as follows:

1. OTT command line
2. Value in INTYPE file
3. User-defined configuration file
4. Default configuration file

For global options (that is, options on the command line or options at the beginning of the INTYPE file before any TYPE statements), the value on the command line overrides the value in the INTYPE file. (The options that can be specified globally in the INTYPE file are CASE, INITFILE, INITFUNC, MAPFILE and MAPFUNC, but not HFILE or CPPFILE.) Anything in the INTYPE file in a TYPE specification applies to a particular type only and overrides anything on the command line that would otherwise apply to the type. So if you enter TYPE person HFILE=p.h, then it applies to person only and overrides the HFILE on the command line. The statement is not considered a command line parameter.

Setting Parameters on the Command Line
Parameters (also called options) set on the command line override any parameters or option set elsewhere.

Setting Parameters in the INTYPE File
The INTYPE file gives a list of types for the OTT utility to translate.

The parameters CASE, CPPFILE, HFILE, INITFILE, INITFUNC, MAPFILE, and MAPFUNC can appear in the INTYPE file.

Setting Parameters in the Configuration File
A configuration file is a text file that contains OTT parameters. Each nonblank line in the file contains one parameter, with its associated value or values. If multiple parameters are on the same line, then only the first one is used. No blank space is allowed on any nonblank line of a configuration file.

A configuration file can be named on the command line. In addition, a default configuration file is always read. This default configuration file must always exist, but can be empty. The name of the default configuration file is ottcfg.cfg, and the location of the file is operating system-specific.

See Also: Your operating system-specific documentation for more information about the location of the default configuration file.

Invoking the OTT Utility on the Command Line
On most platforms, the OTT utility is invoked on the command line. You can specify the input and output files and the database connection information at the command line, among other things.
Invoking the OTT Utility

See Also: Your operating system-specific documentation to see how to invoke the OTT utility on your operating system

Example 8–3 How to Invoke the OTT Utility to Generate C++ Classes

OTT attempts to connect with username demousr; the system prompts for the password.

```
ott userid=demousr intype=demoin.typ outtype=demoout.typ code=cpp
  hfile=demo.h cppfile=demo.cpp mapfile=RegisterMappings.cpp
```

Caution: No spaces are permitted around the equals sign (=) on the OTT command line.

An OTT command line statement consists of the command OTT, followed by a list of OTT utility parameters.

The HFILE parameter is almost always used. If omitted, then HFILE must be specified individually for each type in the INTYPE file. If the OTT utility determines that a type not listed in the INTYPE file must be translated, then an error is reported. Therefore, it is safe to omit the HFILE parameter only if the INTYPE file was previously generated as an OTT OUTTYPE file.

If the INTYPE file is omitted, then the entire schema is translated. See the parameter descriptions in the following section for more information.

Elements Used on the OTT Command Line

Elements used on the OTT command line are:

- OTT command that invokes the OTT utility. It must be the first item on the command line.
- USERID parameter on page 8-11
- INTYPE parameter on page 8-7
- OUTTYPE parameter on page 8-7.
- CODE parameter on page 8-6.
- HFILE parameter on page 8-7.
- CPPFILE parameter on page 8-6.
- MAPFILE parameter on page 8-7.

OTT Utility Parameters

To generate C++ using the OTT utility, the CODE parameter must be set to CODE=CPP. Once CODE=CPP is specified, you are required to specify the CPPFILE and MAPFILE parameters to define the filenames for the method implementation file and the mappings registration function file. The name of the mapping function is derived by the OTT utility from the MAPFILE or you may specify the name with the MAPFUNC parameter. ATTRACCESS is also an optional parameter that can be specified to change the generated code. These parameters control the generation of C++ classes.

- Enter parameters on the OTT command line where parameter is the literal parameter string and value is a valid parameter setting. The literal parameter string is not case sensitive:

```
parameter=value
```
Separate command line parameters by using either spaces or tabs.

Parameters can also appear within a configuration file, but, in that case, no whitespace is permitted within a line, and each parameter must appear on a separate line. Additionally, the parameters CASE, CPPFILE, HFILE, INITFILE, INTFUNC, MAPFILE, and MAPFUNC can appear in the INTYPE file.

Table 8–1 lists all OTT Utility parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTRACCESS</td>
<td>Specifies whether the access to type attributes is PROTECTED or PRIVATE.</td>
</tr>
<tr>
<td>CASE</td>
<td>Affects the letter case of generated C++ identifiers</td>
</tr>
<tr>
<td>CODE</td>
<td>Specifies the target language for the translation. Use CPP.</td>
</tr>
<tr>
<td>CONFIG</td>
<td>Specifies the name of the OTT configuration file that lists commonly used parameter specifications.</td>
</tr>
<tr>
<td>CPPFILE</td>
<td>Specifies the name of the C++ source file into which the method implementations are written.</td>
</tr>
<tr>
<td>ERRTYPE</td>
<td>Specifies the name of the error message output file.</td>
</tr>
<tr>
<td>HFILE</td>
<td>Specifies the name of the C++ header file to which the generated C++ classes are written.</td>
</tr>
<tr>
<td>INTYPE</td>
<td>Specifies the name of the INTYPE file.</td>
</tr>
<tr>
<td>MAPFILE</td>
<td>Specifies the name of the mapping file and the corresponding header file generated by the OTT utility.</td>
</tr>
<tr>
<td>MAPFUNC</td>
<td>Specifies the name of the function used to register generated mappings.</td>
</tr>
<tr>
<td>OUTTYPE</td>
<td>Specifies the name of the OUTTYPE file.</td>
</tr>
<tr>
<td>SCHEMA_NAMES</td>
<td>Controls the qualifying the database name of a type from the default schema</td>
</tr>
<tr>
<td>TRANSITIVE</td>
<td>Indicates whether to translate type dependency that are not explicitly listed in the INTYPE.</td>
</tr>
<tr>
<td>UNICODE</td>
<td>Indicates whether the application should provide UTF16 support generate UString types.</td>
</tr>
<tr>
<td>USE_MARKER</td>
<td>Indicates whether OTT markers should be supported to carry forward user added cod</td>
</tr>
<tr>
<td>USERID</td>
<td>Specifies the database connection information that the OTT utility uses.</td>
</tr>
</tbody>
</table>

**ATTRACCESS**

This parameter specifies access to type attributes:

- PROTECTED is the default.
- PRIVATE indicates that the OTT utility generates accessory and mutator methods for each type attribute, getXXX() and setXXX().

**CASE**

This parameter affects the letter case of generated C++ identifiers. The valid values of CASE are:
Invoking the OTT Utility

- **SAME** is the case of letters remains unchanged when converting database type and attribute names to C++ identifiers.
- **LOWER** indicates that all uppercase letters are converted to lowercase.
- **UPPER** indicates that all lowercase letters are converted to uppercase.
- **OPPOSITE** indicates that all uppercase letters are converted to lowercase, and all lowercase letters are converted to uppercase.

This parameter affects only those identifiers (attributes or types not explicitly listed) not mentioned in the **INTYPE** file. Case conversion takes place after a legal identifier has been generated.

Case insensitive SQL identifiers not mentioned in the **INTYPE** file appear in uppercase if **CASE=SAME**, and in lowercase if **CASE=OPPOSITE**. A SQL identifier is case insensitive if it was not quoted when it was declared.

**CODE**

This parameter specifies the host language to be output by the OTT utility. **CODE=CPP** must be specified for the OTT utility to generate C++ code for OCCI applications.

**CONFIG**

This parameter specifies the name of the OTT configuration file that lists commonly used parameter specifications. Parameter specifications are also read from a system configuration file found in an operating system-dependent location. All remaining parameter specifications must appear either on the command line or in the **INTYPE** file.

The **CONFIG** parameter can only be specified on the OTT command line. It is not allowed in the **CONFIG** file.

**CPPFILE**

This parameter specifies the name of the C++ source file that contains the method implementations generated by the OTT utility. The methods generated in this file are called by OCCI while instantiating the objects and are not to be called directly in the application.

This parameter is required under the following conditions:

- A type not mentioned in the **INTYPE** file must be generated and two or more **CPPFILE**s are being generated. In this case, the unmentioned type goes in the **CPPFILE** specified on the command line.
- The **INTYPE** parameter is not specified, and you want the OTT utility to translate all the types in the schema.

This parameter is optional when the **CPPFILE** is specified for individual types in the **INTYPE** file.

**ERRTYPE**

This parameter specifies the name of the error message output file. Information and error messages are sent to the standard output regardless of whether the **ERRTYPE** parameter is specified. Essentially, the **ERRTYPE** file is a copy of the **INTYPE** file with error messages added. In most cases, an error message includes a pointer to the text that caused the error.
If the filename specified for the `ERRTYPE` parameter on the command line does not include an extension, a platform-specific extension, like `.TLS` or `.tls`, is added automatically.

**HFILE**
This parameter specifies the name of the header (.h) file to be generated by the OTT utility. The `HFILE` specified on the command line contains the declarations of types that are mentioned in the `INTYPE` file but whose header files are not specified there.

This parameter is required unless the header file for each type is specified individually in the `INTYPE` file. This parameter is also required if a type not mentioned in the `INTYPE` file must be generated because other types require it, and these other types are declared in two or more different files.

If the filename specified for the `HFILE` parameter on the command line or in the `INTYPE` file does not include an extension, a platform-specific extension, like `.H` or `.h`, is added automatically.

**INTYPE**
This parameter specifies the name of the file from which to read the list of object type specifications. The OTT utility translates each type in the list. If the `INTYPE` parameter is not specified, all types in the user's schema is translated.

If the filename specified for the `INTYPE` parameter on the command line does not include an extension, a platform-specific extension, like `.TYP` or `.typ`, is automatically added.

`INTYPE=` may be omitted if `USERID` and `INTYPE` are the first two parameters, in that order, and `USERID=` is omitted.

The `INTYPE` file can be thought of as a makefile for type declarations. It lists the types for which C++ classes are needed.

See Also:  "Structure of the INTYPE File" on page 8-13 for more information about the format of the `INTYPE` file

**MAPFILE**
This parameter specifies the name of the mapping file (XXX.cpp) and corresponding header file (XXX.h) that are generated by the OTT utility. The XXX.cpp file contains the implementation of the functions to register the mappings, while the XXX.h file contains the prototype for the function.

This parameter may be specified either on the command line or in the `INTYPE` file.

**MAPFUNC**
This parameter specifies the name of the function to be used to register the mappings generated by the OTT utility.

If this parameter is omitted, then the name of the function to register the mappings is derived from the filename specified in the `MAPFILE` parameter.

This parameter may be specified either on the command line or in the `INTYPE` file.

**OUTTYPE**
This parameter specifies the name of the file into which the OTT utility writes type information for all the object data types it processes. This file includes all types explicitly named in the `INTYPE` file, and may include additional types that are
translated because they are used in the declarations of other types that must be translated. This file may be used as an INTYPE file in a future invocation of the OTT utility.

If the INTYPE and OUTTYPE parameters refer to the same file, then the new INTYPE information replaces the old information in the INTYPE file. This provides a convenient way for the same INTYPE file to be used repeatedly in the cycle of altering types, generating type declarations, editing source code, precompiling, compiling, and debugging.

If the filename specified for the OUTTYPE parameter on the command line or in the INTYPE file does not include an extension, a platform-specific extension, like .TYP or .typ, is automatically added.

**SCHEMA_NAMES**

This parameter offers control in qualifying the database name of a type from the default schema that is named in the OUTTYPE file. The OUTTYPE file generated by the OTT utility contains information about the types processed by the OTT utility, including the type names. Valid values include:

- ** ALWAYS** (default) indicates that all type names in the OUTTYPE file are qualified with a schema name.
- ** IF_NEEDED** indicates that the type names in the OUTTYPE file that belong to the default schema are not qualified with a schema name. Type names belonging to other schemas are qualified with the schema name.
- ** FROM_INTYPE** indicates that a type mentioned in the INTYPE file is qualified with a schema name in the OUTTYPE file only if it was qualified with a schema name in the INTYPE file. A type in the default schema that is not mentioned in the INTYPE file but generated because of type dependency is written with a schema name only if the first type encountered by the OTT utility that depends on it is also written with a schema name. However, a type that is not in the default schema to which the OTT utility is connected is always written with an explicit schema name.

The name of a type from a schema other that the default schema is always qualified with a schema name in the OUTTYPE file.

The schema name, or its absence, determines in which schema the type is found during program execution.

**Example 8–4  How to use the SCHEMA_NAMES Parameter in OTT Utility**

Consider an example where the SCHEMA_NAMES parameter is set to FROM_INTYPE, and the INTYPE file contains the following:

```
TYPE Person
TYPE joe.Dept
TYPE sam.Company
```

If the OTT utility and the application both connect to schema joe, then the application uses the same type (joe.Person) that the OTT utility uses. If the OTT utility connects to schema joe but the application connects to schema mary, then the application uses the type mary.Person. This behavior is appropriate only if the same CREATE TYPE Person statement has been executed in schema joe and schema mary.

On the other hand, the application uses type joe.Dept regardless of which schema the application is connected to. If this is the behavior you want, then be sure to include schema names with your type names in the INTYPE file.
In some cases, the OTT utility translates a type that the user did not explicitly name. For example, consider the following SQL declarations:

```sql
CREATE TYPE Address AS OBJECT
{
    street VARCHAR2(40),
    city VARCHAR(30),
    state CHAR(2),
    zip_code CHAR(10)
};

CREATE TYPE Person AS OBJECT
{
    name CHAR(20),
    age NUMBER,
    addr ADDRESS
};
```

Suppose that the OTT utility connects to schema `joe`, `SCHEMA_NAMES=FROM_INTYPE` is specified, and the user's `INTYPE` files include either `TYPE Person` or `TYPE joe.Person`. The `INTYPE` file does not mention the type `joe.Address`, which is used as a nested object type in type `joe.Person`.

- If `TYPE Person` appears in the `INTYPE` file, then `TYPE Person` and `TYPE Address` appears in the `OUTTYPE` file.
- If the `joe.Address` type is embedded in several types translated by the OTT utility, but it is not explicitly mentioned in the `INTYPE` file, then the decision of whether to use a schema name is made the first time the OTT utility encounters the embedded `joe.Address` type. If, for some reason, the user wants type `joe.Address` to have a schema name but does not want type `Person` to have one, then you must explicitly request this in the `INTYPE` file: `TYPE joe.Address`.

In the usual case in which each type is declared in a single schema, it is safest for you to qualify all type names with schema names in the `INTYPE` file.

**TRANSITIVE**

This parameter indicates whether type dependencies not explicitly listed in the `INTYPE` file are to be translated. Valid values are:

- **TRUE** (default): types needed by other types and not mentioned in the `INTYPE` file are generated
- **FALSE**: types not mentioned in the `INTYPE` file are not generated, even if they are used as attribute types of other generated types.

**UNICODE**

This parameter specifies whether the application provides unicode (UTF16) support.

- **NONE** (default)
- **ALL**: All CHAR (CHAR/VARCHAR) and NCHAR (NCHAR/NVARCHAR2) type attributes are declared as UString type in the OTT generated C++ class files. The corresponding getXXX() / setXXX() return values or parameters are UString types. The generated persistent operator new would also take only UString arguments.
This setting is necessary when both the client characterset and the national characterset is UTF16.

- **ONLYNCHAR**: Similar to the ALL option, but only NCHAR type attributes are declared as UString.

  This setting is necessary when the application sets only the Environment’s national characterset to UTF16.

**Example 8–5  How to Define a Schema for Unicode Support in OTT**

```sql
create type CitiesList as varray(100) of varchar2(100);
create type Country as object
(CNo Number(10),
 CName Varchar2(100),
 CNationalName NVarchar2(100),
 MainCities CitiesList);
```

**Example 8–6  How to Use UNICODE=ALL Parameter in OTT**

```cpp
class Country : public oracle::occi::PObject
{
private:
    oracle::occi::Number CNO;
    oracle::occi::UString CNAME;
    oracle::occi::UString CNATIONALNAME;
    OCCI_STD_NAMESPACE::vector< oracle::occi::UString > MAINCITIES;

public:

    oracle::occi::Number getCno() const;
    void setCno(const oracle::occi::Number &value);

    oracle::occi::UString getCname() const;
    void setCname(const oracle::occi::UString &value);

    oracle::occi::UString getCnationalname() const;
    void setCnationalname(const oracle::occi::UString &value);

    OCCI_STD_NAMESPACE::vector< oracle::occi::UString >& getMaincities();
    const OCCI_STD_NAMESPACE::vector< oracle::occi::UString >&
    getMaincities() const;
    void setMaincities(const OCCI_STD_NAMESPACE::vector< oracle::occi::UString
    >& &value);
    ...
}
```

**Example 8–7  How to Use UNICODE=ONLYCHAR Parameter in OTT**

```cpp
class Country : public oracle::occi::PObject
{
private:
    oracle::occi::Number CNO;
    oracle::occi::String CNAME;
    oracle::occi::UString CNATIONALNAME;
    OCCI_STD_NAMESPACE::vector< std::string > MAINCITIES;

public:

    oracle::occi::Number getCno() const;
```
void setCno(const oracle::occi::Number &value);

oracle::occi::string getCname() const;
void setCname(const OCCI_STD_NAMESPACE::string &value);

oracle::occi::UString getCNationalname() const;
void setCNationalname(const oracle::occi::UString &value);

OCCI_STD_NAMESPACE::vector< OCCI_STD_NAMESPACE::string >&
getMaincities();
const OCCI_STD_NAMESPACE::vector< OCCI_STD_NAMESPACE::string >&
getMaincities() const;
void setMaincities(const OCCI_STD_NAMESPACE::vector
< OCCI_STD_NAMESPACE::string >& value);

...

USE_MARKER

This parameter indicates whether to support OTT markers for carrying forward user
added code. Valid values are:

■ FALSE (default): User-supplied code is not carried forward, even if the code is
added between OTT_USERCODE_START and OTT_USERCODE_END markers.

■ TRUE: User-supplied code, between the markers OTT_USER_CODESTART and
OTT_USERCODE_END, is carried forward when the same file is generated again.

USERID

This parameter specifies the Oracle username and optional database name (Oracle Net
database specification string). If the database name is omitted, the default database is
assumed.

USERID=username[@db_name]

If this is the first parameter, then USERID= may be omitted as shown:

OTT username ...

Note that the system prompts you for the password that corresponds to the user id.
This parameter is optional. If omitted, the OTT utility automatically attempts to
connect to the default database as user OPS$username, where username is the user's
operating system username.

Where OTT Parameters Can Appear

Supply OTT parameters on the command line, in a CONFIG file named on the
command line, or both. Some parameters are also allowed in the INTYPE file.

The OTT utility is invoked as follows:

OTT parameters

You can name a configuration file on the command line with the CONFIG parameter as
follows:

CONFIG=filename

If you name this parameter on the command line, then additional parameters are read
from the configuration file named filename.
In addition, parameters are also read from a default configuration file that resides in an operating system-dependent location. This file must exist, but can be empty. If you choose to enter data in the configuration file, note that no white space is allowed on a line and parameters must be entered one to a line.

If the OTT utility is executed without any arguments, then an online parameter reference is displayed.

The types for the OTT utility to translate are named in the file specified by the INTYPE parameter. The parameters CASE, CPPFILE, HFILE, INITFILE, INITFUNC, MAPFILE, and MAPFUNC may also appear in the INTYPE file. OUTTYPE files generated by the OTT utility include the CASE parameter, and include the INITFILE, and INITFUNC parameters if an initialization file was generated or the MAPFILE and MAPFUNC parameters if C++ codes was generated. The OUTTYPE file and the CPPFILE for C++ specify the HFILE individually for each type.

The case of the OTT command is operating system-dependent.

File Name Comparison Restriction

Currently, the OTT utility determines if two files are the same by comparing the filenames provided by the user either on the command line or in the INTYPE file. But one potential problem can occur when the OTT utility must know if two filenames refer to the same file. For example, if the OTT-generated file foo.h requires a type declaration written to fool.h, and another type declaration written to /private/smith/fool.h, then the OTT utility should generate one #include if the two files are the same, and two #includes if the files are different. In practice, though, it concludes that the two files are different, and generates two #includes as follows:

```c
#ifndef FOO1_ORACLE
#include "foo1.h"
#endif
#ifndef FOO1_ORACLE
#include "/private/smith/fool.h"
#endif
```

If foo1.h and /private/smith/fool.h are different files, then only the first one is included. If foo1.h and /private/smith/fool.h are the same file, then a redundant #include is written.

Therefore, if a file is mentioned several times on the command line or in the INTYPE file, then each mention of the file should use the same filename.

Using the INTYPE File

When you run the OTT utility, the INTYPE file tells the OTT utility which database types should be translated. The INTYPE file also controls the naming of the generated structures or classes. You can either create an INTYPE file or use the OUTTYPE file of a previous invocation of the OTT utility. If you do not use an INTYPE file, then all types in the schema to which the OTT utility connects are translated.

Overview of the INTYPE File

**Example 8–8  How to Create a User Defined INTYPE File Using the OTT Utility**

```c
CASE=LOWER
TYPE employee
```

8-12  Oracle C++ Call Interface Programmer's Guide
TRANSLATE SALARY$ AS salary
  DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o

- In the first line, the **CASE** parameter indicates that generated C identifiers should be in lowercase. However, this **CASE** parameter is only applied to those identifiers that are not explicitly mentioned in the INTYPE file. Thus, `employee` and `ADDRESS` would always result in C structures `employee` and `ADDRESS`, respectively. The members of these structures are named in lowercase.

- The lines that begin with the **TYPE** keyword specify which types in the database should be translated. In this case, the `EMPLOYEE`, `ADDRESS`, `ITEM`, `PERSON`, and `PURCHASE_ORDER` types are set to be translated.

- The **TRANSLATE...AS** keywords specify that the name of an object attribute should be changed when the type is translated into a C structure. In this case, the `SALARY$` attribute of the `employee` type is translated to `salary`.

- The **AS** keyword in the final line specifies that the name of an object type should be changed when it is translated into a structure. In this case, the `purchase_order` database type is translated into a structure called `p_o`.

The OTT utility may have to translate additional types that are not listed in the INTYPE file. This is because the OTT utility analyzes the types in the INTYPE file for type dependencies before performing the translation, and it translates other types as necessary. For example, if the `ADDRESS` type were not listed in the INTYPE file, but the `Person` type had an attribute of type `ADDRESS`, then the OTT utility would still translate `ADDRESS` because it is required to define the `Person` type.

You may indicate whether the OTT utility should generate required object types that are not specified in the INTYPE file. Set `TRANSITIVE=FALSE` so the OTT utility does not generate required object types. The default is `TRANSITIVE=TRUE`.

A normal case insensitive SQL identifier can be spelled in any combination of uppercase and lowercase in the INTYPE file, and is not quoted.

Use quotation marks, such as `TYPE "Person"` to reference SQL identifiers that have been created in a case sensitive manner, for example, `CREATE TYPE "Person"`. A SQL identifier is case sensitive if it was quoted when it was declared. Quotation marks can also be used to refer to a SQL identifier that is an OTT-reserved word, for example, `TYPE "CASE"`. In this case, the quoted name must be in uppercase if the SQL identifier was created in a case insensitive manner, for example, `CREATE TYPE Case`. If an OTT-reserved word is used to refer to the name of a SQL identifier but is not quoted, then the OTT utility reports a syntax error in the INTYPE file.

**See Also:**
- "Structure of the INTYPE File" on page 8-13 for a more detailed specification of the structure of the INTYPE file and the available options.
- "CASE" on page 8-5 for further information regarding the **CASE** parameter

**Structure of the INTYPE File**

The INTYPE and OUTTYPE files list the types translated by the OTT utility and provide all the information needed to determine how a type or attribute name is translated to a
Using the INTYPE File

A legal C or C++ identifier. These files contain one or more type specifications, and may also contain specifications of CASE, CPPFILE, HFILE, INITFILE, INITFUNC, MAPFILE, or MAPFUNC.

If the CASE, INITFILE, INITFUNC, MAPFILE, or MAPFUNC options are present, then they must precede any type specifications. If these options appear both on the command line and in the INTYPE file, then the value on the command line is used.

See Also: "Overview of the OUTTYPE File" on page 8-23 for an example of a simple user-defined INTYPE file and of the full OUTTYPE file that the OTT utility generates from it

INTYPE File Type Specifications

A type specification in the INTYPE file names an object data type that is to be translated. The following is an example of a user-created INTYPE file:

```
TYPE employee
    TRANSLATE SALARY$ AS salary
    DEPTNO AS department
TYPE ADDRESS
TYPE PURCHASE_ORDER AS p_o
```

The structure of a type specification is as follows:

```
TYPE type_name
    [GENERATE type_identifier]
    [AS type_identifier]
    [VERSION [=] version_string]
    [HFILE [=] hfile_name]
    [CPPFILE [=] cppfile_name]
    [TRANSLATE{member_name [AS identifier]}...]
```

The type_name syntax follows this form:

```
[schema_name.]type_name
```

In this syntax, schema_name is the name of the schema that owns the given object data type, and type_name is the name of the type. The default schema, if one is not specified, is that of the userID invoking the OTT utility. To use a specific schema, you must use schema_name.

The components of the type specification are:

- **type_name**: Name of the object data type.
- **type_identifier**: C / C++ identifier used to represent the class. The GENERATE clause is used to specify the name of the class that the OTT utility generates. The AS clause specifies the name of the class that you write. The GENERATE clause is typically used to extend a class. The AS clause, when optionally used without the GENERATE clause, specifies the name of the C structure or the C++ class that represents the user-defined type.
- **version_string**: Version string of the type that was used when the code was generated by the previous invocation of the OTT utility. The version string is generated by the OTT utility and written to the OUTTYPE file, which can later be used as the INTYPE file in later invocations of the OTT utility. The version string does not affect how the OTT utility operates, but can be used to select which version of the object data type is used in the running program.
Using the INTYPE File

- **hfile_name**: Name of the header file into which the declarations of the corresponding class are written. If you omit the HFILE clause, then the file specified by the command line HFILE parameter is used.

- **cppfile_name**: Name of the C++ source file into which the method implementations of the corresponding class is written. If you omit the CPPFILE clause, the file specified by the command line CPPFILE parameter is used.

- **member_name**: Name of an attribute (data member) that is to be translated to the identifier.

- **identifier**: C / C++ identifier used to represent the attribute in the program. You can specify identifiers in this way for any number of attributes. The default name mapping algorithm is used for the attributes not mentioned.

An object data type may be translated for one of two reasons:

- It appears in the INTYPE file.

- It is required to declare another type that must be translated, and the TRANSITIVE parameter is set to TRUE.

If a type that is not mentioned explicitly is necessary to types declared in exactly one file, then the translation of the required type is written to the same files as the explicitly declared types that require it.

If a type that is not mentioned explicitly is necessary to types declared in multiple files, then the translation of the required type is written to the global HFILE file.

You may indicate whether the OTT utility should generate required object types that are not specified in the INTYPE file. Set TRANSITIVE=FALSE so the OTT utility does not to generate required object types. The default is TRANSITIVE=TRUE.

**Nested #include File Generation**

HFILE files generated by the OTT utility #include other necessary files, and #define a symbol constructed from the name of the file. This symbol #define can then be used to determine if the related HFILE file has been included. Consider, for example, a database with the following types:

```sql
create type px1 AS OBJECT (col1 number, col2 integer);
cREATE TYPE px2 AS OBJECT (col1 px1);
cREATE TYPE px3 AS OBJECT (col1 px1);
```

The INTYPE file contains the following information:

```text
CASE=lower
type px1
  hfile tott95a.h
type px3
  hfile tott95b.h
```

You invoke the OTT utility as follows:

```bash
>ott hr intype=tott95i.typ outtype=tott95o.typ code=cpp ...
Enter password: password
```

The OTT utility then generates the following two header files, named tott95a.h and tott95b.h. They are listed in
Example 8–9  Listing of ott95a.h

```c++
#ifndef TOTT95A_ORACLE
#define TOTT95A_ORACLE
#endif

#ifndef OCCI_ORACLE
#define OCCI_ORACLE
#endif

#include <occi.h>

/*********************************************************************
// generated declarations for the PX1 object type.
*********************************************************************/

class px1 : public oracle::occi::PObject {

protected:
    oracle::occi::Number col1;
    oracle::occi::Number col2;

public:
    void *operator new(size_t size);
    void *operator new(size_t size, const oracle::occi::Connection * sess,
                       const OCCI_STD_NAMESPACE::string & table);
    void *operator new(size_t, void *ctxOCCI_);
    void *operator new(size_t size, const oracle::occi::Connection * sess,
                       const OCCI_STD_NAMESPACE::string & tableName,
                       const OCCI_STD_NAMESPACE::string & typename,
                       const OCCI_STD_NAMESPACE::string & tableSchema,
                       const OCCI_STD_NAMESPACE::string & typeSchema);
    void getSQLTypeName(oracle::occi::Environment * env, void ** schemaName,
                          unsigned int & schemaNameLen, void ** typeName,
                          unsigned int & typeNameLen) const;
    px1();
    px1(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
    static void *readSQL(void *ctxOCCI_);
    virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
    static void writeSQL(void *objOCCI_, void *ctxOCCI_);
    virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
    ~px1();
};
#endif
```

Example 8–10  Listing of ott95b.h

```c++
#ifndef TOTT95B_ORACLE
#define TOTT95B_ORACLE
#endif

#ifndef OCCI_ORACLE
#define OCCI_ORACLE
#endif

#ifndef TOTT95A_ORACLE
#define TOTT95A_ORACLE
#endif

#include <occi.h>

/*********************************************************************
// generated declarations for the PX3 object type.
*********************************************************************/

class px3 : public oracle::occi::PObject {

protected:
    oracle::occi::Number col1;
    oracle::occi::Number col2;

public:
    void *operator new(size_t size);
    void *operator new(size_t size, const oracle::occi::Connection * sess,
                       const OCCI_STD_NAMESPACE::string & table);
    void *operator new(size_t, void *ctxOCCI_);
    void *operator new(size_t size, const oracle::occi::Connection * sess,
                       const OCCI_STD_NAMESPACE::string & tableName,
                       const OCCI_STD_NAMESPACE::string & typeName,
                       const OCCI_STD_NAMESPACE::string & tableSchema,
                       const OCCI_STD_NAMESPACE::string & typeSchema);
    void getSQLTypeName(oracle::occi::Environment * env, void ** schemaName,
                          unsigned int & schemaNameLen, void ** typeName,
                          unsigned int & typeNameLen) const;
    px3();
    px3(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
    static void *readSQL(void *ctxOCCI_);
    virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
    static void writeSQL(void *objOCCI_, void *ctxOCCI_);
    virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
    ~px3();
};
```
protected:
    px1 * col1;

public:
    void *operator new(size_t size);
    void *operator new(size_t size, const oracle::occi::Connection * sess,
                        const OCCI_STD_NAMESPACE::string& table);
    void *operator new(size_t, void *ctxOCCI_);
    void *operator new(size_t size, const oracle::occi::Connection *sess,
                        const OCCI_STD_NAMESPACE::string &tableName,
                        const OCCI_STD_NAMESPACE::string &typeName,
                        const OCCI_STD_NAMESPACE::string &tableSchema,
                        const OCCI_STD_NAMESPACE::string &typeSchema);
    void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
                        unsigned int &schemaNameLen, void **typeName,
                        unsigned int &typeNameLen) const;
    px3();
    px3(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
    static void *readSQL(void *ctxOCCI_);
    virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
    static void writeSQL(void *objOCCI_, void *ctxOCCI_);
    virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
    ~px3();

#endif

In the tott95b.h file, the symbol TOTT95B_ORACLE is #define d at the beginning
of the file. This enables you to conditionally #include this header file in another file,
using the following construct:

#ifndef TOTT95B_ORACLE
#include "tott95b.h"
#endif

By using this technique, you can #include tott95b.h in, say foo.h, without
having to know whether some other file #included in foo.h also #includes
tott95b.h.

Next, the file tott95a.h is included because it contains the declaration of struct
px1, that tott95b.h requires. When the INTYPE file requests that type declarations
be written to multiple files, the OTT utility determines which other files each HFILE
must #include, and generates each necessary #include.

Note that the OTT utility uses quotes in this #include. When a program including
tott95b.h is compiled, the search for tott95a.h begins where the source program
was found, and thereafter follows an implementation-defined search rule. If
tott95a.h cannot be found in this way, then a complete filename (for example, a
UNIX absolute path name beginning with a slash character (/)) is necessary in the
INTYPE file to specify the location of tott95a.h.

**OTT Utility Data Type Mappings**

When the OTT utility generates a C++ class from a database type, the structure or class
contains one element corresponding to each attribute of the object type. The data types
of the attributes are mapped to types that are used in Oracle object data types. The
data types found in Oracle include a set of predefined, primitive types and provide for
the creation of user-defined types, like object types and collections.
The set of predefined types includes standard types that are familiar to most programmers, including number and character types. It also includes large object data types (for example, BLOB or CLOB).

**Example 8–11  How to Represent Object Attributes Using the OTT Utility**

Oracle also includes a set of predefined types that are used to represent object type attributes in C++ classes. Consider the following object type definition, and its corresponding OTT-generated structure declarations:

```sql
CREATE TYPE employee AS OBJECT
    ( name       VARCHAR2(30),
      empno      NUMBER,
      deptno     NUMBER,
      hiredate   DATE,
      salary     NUMBER
    );
```

The OTT utility, assuming that the `CASE` parameter is set to `LOWER` and there are no explicit mappings of type or attribute names, produces the following output:

```c++
#ifndef DATATYPES_ORACLE
#define DATATYPES_ORACLE
#ifndef OCCI_ORACLE
#include <occi.h>
#endif
/****************************/
// generated declarations for the EMPLOYEE object type.
/****************************/

class employee : public oracle::occi::PObject {

protected:
    OCCI_STD_NAMESPACE::string NAME;
    oracle::occi::Number EMPNO;
    oracle::occi::Number DEPTNO;
    oracle::occi::Date HIREDATE;
    oracle::occi::Number SALARY;

public:
    void *operator new(size_t size);
    void *operator new(size_t size, const oracle::occi::Connection * sess,
        const OCCI_STD_NAMESPACE::string& table);
    void *operator new(size_t size, void *ctxOCCI_);
    void *operator new(size_t size, const oracle::occi::Connection *sess,
        const OCCI_STD_NAMESPACE::string &tableName,
        const OCCI_STD_NAMESPACE::string &typeName,
        const OCCI_STD_NAMESPACE::string &tableSchema,
        const OCCI_STD_NAMESPACE::string &typeSchema);
    void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
        unsigned int &schemaNameLen, void **typeName,
        unsigned int &typeNameLen) const;
    employee();
    employee(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { }; static void *readSQL(void *ctxOCCI_);
    virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
    static void writeSQL(void *objOCCI_, void *ctxOCCI_);
    virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);~employee();
```
Table 8–2 lists the mappings from types that can be used as attributes to object data types that are generated by the OTT utility.

### Table 8–2 C++ Object Data Type Mappings for Object Type Attributes

<table>
<thead>
<tr>
<th>Object Attribute Types</th>
<th>C++ Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFILE</td>
<td>Bfile</td>
</tr>
<tr>
<td>BLOB</td>
<td>Blob</td>
</tr>
<tr>
<td>BINARY_DOUBLE</td>
<td>BDouble</td>
</tr>
<tr>
<td>BINARY_FLOAT</td>
<td>BFloat</td>
</tr>
<tr>
<td>CHAR(n), CHARACTER(n)</td>
<td>string</td>
</tr>
<tr>
<td>CLOB</td>
<td>Clob</td>
</tr>
<tr>
<td>DATE</td>
<td>Date</td>
</tr>
<tr>
<td>DEC, DEC(n), DEC(n,n)</td>
<td>Number</td>
</tr>
<tr>
<td>DECIMAL, DECIMAL(n), DECIMAL(n,n)</td>
<td>Number</td>
</tr>
<tr>
<td>FLOAT, FLOAT(n), DOUBLE PRECISION</td>
<td>Number</td>
</tr>
<tr>
<td>INT, INTEGER, SMALLINT</td>
<td>Number</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td>IntervalDS</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>IntervalYM</td>
</tr>
<tr>
<td>Nested Object Type</td>
<td>C++ name of the nested object type</td>
</tr>
<tr>
<td>NESTED TABLE</td>
<td>vector&lt;attribute_type&gt;</td>
</tr>
<tr>
<td>NUMBER, NUMBER(n), NUMBER(n,n)</td>
<td>Number</td>
</tr>
<tr>
<td>NUMERIC, NUMERIC(n), NUMERIC(n,n)</td>
<td>Number</td>
</tr>
<tr>
<td>RAW</td>
<td>Bytes</td>
</tr>
<tr>
<td>REAL</td>
<td>Number</td>
</tr>
<tr>
<td>REF</td>
<td>Ref&lt;attribute_type&gt;</td>
</tr>
<tr>
<td>TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE</td>
<td>Timestamp</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>string</td>
</tr>
<tr>
<td>VARCHAR2(n)</td>
<td>string</td>
</tr>
<tr>
<td>VARRAY</td>
<td>vector&lt;attribute_type&gt;</td>
</tr>
</tbody>
</table>

**Example 8–12  How to Map Object Data Types Using the OTT Utility**

The example assumes that the following database types are created:

```sql
CREATE TYPE my_varray AS VARRAY(5) of integer;
CREATE TYPE object_type AS OBJECT
   (object_name VARCHAR2(20));
CREATE TYPE other_type AS OBJECT
   (object_number NUMBER);
```
CREATE TYPE my_table AS TABLE OF object_type;

CREATE TYPE many_types AS OBJECT
{
    the_varchar VARCHAR2(30),
    the_char CHAR(3),
    the_blob BLOB,
    the_clob CLOB,
    the_object object_type,
    another_ref REF other_type,
    the_ref REF many_types,
    the_varray my_varray,
    the_table my_table,
    the_date DATE,
    the_num NUMBER,
    the_raw RAW(255)
};

An INTYPE file exists, and includes the following:

CASE = LOWER
TYPE many_types

The following is an example of the OTT type mappings for C++, given the types created in the example in the previous section, and an INTYPE file that includes the following:

CASE = LOWER
TYPE many_types

#ifndef MYFILENAME_ORACLE
#define MYFILENAME_ORACLE
#endif

#ifndef OCCLI_ORACLE
#include <occi.h>
#endif

/****************************/
//  generated declarations for the OBJECT_TYPE object type.
/****************************/

class object_type : public oracle::occi::PObject
{
    protected:
        OCCI_STD_NAMESPACE::string object_name;

    public:
        void *operator new(size_t size);
        void *operator new(size_t size, const oracle::occi::Connection * sess,
                         const OCCI_STD_NAMESPACE::string& table);
        void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
                            unsigned int &schemaNameLen, void **typeName,
                            unsigned int &typeNameLen) const;
        object_type();
        object_type(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
        static void *readSQL(void *ctxOCCI_);
        virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
        static void writeSQL(void *objOCCI_, void *ctxOCCI_);
        virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
};
/**
 *  generated declarations for the OTHER_TYPE object type.
 */

class other_type : public oracle::occi::PObject
{
    protected:
        oracle::occi::Number object_number;

    public:
        void *operator new(size_t size);
        void *operator new(size_t size, const oracle::occi::Connection * sess,
            const OCCI_STD_NAMESPACE::string& table);
        void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
            unsigned int &schemaNameLen, void **typeName,
            unsigned int &typeNameLen) const;
        other_type();
        other_type(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
        static void *readSQL(void *ctxOCCI_);
        virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
        static void writeSQL(void *objOCCI_, void *ctxOCCI_);
        virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
};

/**
 *  generated declarations for the MANY_TYPES object type.
 */

class many_types : public oracle::occi::PObject
{
    protected:
        OCCI_STD_NAMESPACE::string the_varchar;
        OCCI_STD_NAMESPACE::string the_char;
        oracle::occi::Blob the_blob;
        oracle::occi::Clob the_clob;
        object_type * the_object;
        oracle::occi::Ref< other_type > another_ref;
        oracle::occi::Ref< many_types > the_ref;
        OCCI_STD_NAMESPACE::vector< oracle::occi::Number > the_varray;
        OCCI_STD_NAMESPACE::vector< object_type * > the_table;
        oracle::occi::Date the_date;
        oracle::occi::Number the_num;
        oracle::occi::Bytes the_raw;

    public:
        void *operator new(size_t size);
        void *operator new(size_t size, const oracle::occi::Connection * sess,
            const OCCI_STD_NAMESPACE::string& table);
        void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
            unsigned int &schemaNameLen, void **typeName,
            unsigned int &typeNameLen) const;
        many_types();
        many_types(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
        static void *readSQL(void *ctxOCCI_);
        virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
        static void writeSQL(void *objOCCI_, void *ctxOCCI_);
        virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
};
The OTT utility generates the following C++ class declarations (comments are not part of the OTT output, and are added only to clarify the example):

For C++, when TRANSITIVE=TRUE, the OTT utility automatically translates any types that are used as attributes of a type being translated, including types that are only being accessed by a pointer or REF in an object type attribute. Even though only the many_types object was specified in the INTYPE file for the C++ example, a class declaration was generated for all the object types, including the other_type object, which was only accessed by a REF in the many_types object.

**Default Name Mapping**

When the OTT utility creates a C or C++ identifier name for an object type or attribute, it translates the name from the database character set to a legal C or C++ identifier. First, the name is translated from the database character set to the character set used by the OTT utility. Next, if a translation of the resulting name is supplied in the INTYPE file, that translation is used. Otherwise, the OTT utility translates the name character-by-character to the compiler character set, applying the character case specified in the CASE parameter. The following text describes this in more detail.

When the OTT utility reads the name of a database entity, the name is automatically translated from the database character set to the character set used by the OTT utility. In order for the OTT utility to read the name of the database entity successfully, all the characters of the name must be found in the OTT character set, although a character may have different encodings in the two character sets.

The easiest way to guarantee that the character set used by the OTT utility contains all the necessary characters is to make it the same as the database character set. Note, however, that the OTT character set must be a superset of the compiler character set. That is, if the compiler character set is 7-bit ASCII, then the OTT character set must include 7-bit ASCII as a subset, and if the compiler character set is 7-bit EBCDIC, then the OTT character set must include 7-bit EBCDIC as a subset. The user specifies the character set that the OTT utility uses by setting the NLS_LANG environment variable, or by some other operating system-specific mechanism.

Once the OTT utility has read the name of a database entity, it translates the name from the character set used by the OTT utility to the compiler's character set. If a translation of the name appears in the INTYPE file, then the OTT utility uses that translation.

Otherwise, the OTT utility attempts to translate the name as follows:

1. If the OTT character set is a multibyte character set, all multibyte characters in the name that have single-byte equivalents are converted to those single-byte equivalents.

2. The name is converted from the OTT character set to the compiler character set. The compiler character set is a single-byte character set such as US7ASCII.

3. The case of letters is set according to how the CASE parameter is defined, and any character that is not legal in a C or C++ identifier, or that has no translation in the compiler character set, is replaced by an underscore character (_). If at least one character is replaced by an underscore, then the OTT utility gives a warning message. If all the characters in a name are replaced by underscores, the OTT utility gives an error message.
Character-by-character name translation does not alter underscores, digits, or single-byte letters that appear in the compiler character set, so legal C or C++ identifiers are not altered.

Name translation may, for example, translate accented single-byte characters such as o with an umlaut or an a with an accent grave to o or a, with no accent, and may translate a multibyte letter to its single-byte equivalent. Name translation typically fails if the name contains multibyte characters that lack single-byte equivalents. In this case, the user must specify name translations in the INTYPE file.

The OTT utility does not detect a naming clash caused by two or more database identifiers being mapped to the same C name, nor does it detect a naming problem where a database identifier is mapped to a C keyword.

Overview of the OUTTYPE File

The OUTTYPE file is named on the OTT command line. When the OTT utility generates a C++ header file, it also writes the results of the translation into the OUTTYPE file. This file contains an entry for each of the translated types, including its version string and the header file to which its C++ representation was written.

The OUTTYPE file from one OTT utility run can be used as the INTYPE file for a subsequent invocation of the OTT utility.

Example 8–13 OUTTYPE File Generated by the OTT Utility

In this INTYPE file, the programmer specifies the case for OTT-generated C++ identifiers, and provides a list of types that should be translated. In two of these types, naming conventions are specified. This is what the OUTTYPE file looks like after running the OTT utility:

The following example shows what t:

```plaintext
CASE = LOWER
TYPE EMPLOYEE AS employee
    VERSION = "$8.0"
    HFILE = demo.h
    TRANSLATE SALARY$ AS salary
    DEPTNO AS department
TYPE ADDRESS AS ADDRESS
    VERSION = "$8.0"
    HFILE = demo.h
TYPE ITEM AS item
    VERSION = "$8.0"
    HFILE = demo.h
TYPE "Person" AS Person
    VERSION = "$8.0"
    HFILE = demo.h
TYPE PURCHASE_ORDER AS p_o
    VERSION = "$8.0"
    HFILE = demo.h
```

When examining the contents of the OUTTYPE file, you might discover types listed that were not included in the INTYPE file specification. For example, consider the case where the INTYPE file only specified that the person type was to be translated:

```plaintext
CASE = LOWER
TYPE PERSON
```
If the definition of the person type includes an attribute of type address, then the OUTTYPE file includes entries for both PERSON and ADDRESS. The person type cannot be translated completely without first translating address.

The OTT utility analyzes the types in the INTYPE file for type dependencies before performing the translation, and translates other types as necessary.

You may indicate whether the OTT utility should generate required object types that are not specified in the INTYPE file. Set TRANSITIVE=FALSE so the OTT utility does not generate required object types. The default is TRANSITIVE=TRUE.

The OTT Utility and OCCI Applications

The OTT utility generates objects and maps SQL data types to C++ classes. The OTT utility also implements a few methods called by OCCI when instantiating objects and a function that is called in the OCCI application to register the mappings with the environment. These declarations are stored in a header file that you include (#include) in your OCCI application. The prototype for the function that registers the mappings is written to a separate header file, which you also include in your OCCI application. The method implementations are stored in a C++ source code file (with extension .cpp) that is linked with the OCCI application. The function that registers the mappings is stored in a separate C++ (xxx.cpp) file that is also linked with the application.

Figure 8–1 shows the steps involved in using the OTT utility with OCCI. These steps are described following the figure.

Figure 8–1  The OTT Utility with OCCI

1. Create the type definitions in the database by using the SQL DLL.
2. Create the `INTYPE` file that contains the database types to be translated by the OTT utility.

3. Specify that C++ should be generated and invoke the OTT utility.

   The OTT utility then generates the following files:
   - A header file (with the extension `.h`) that contains C++ class representations of object types; the filename is specified on the OTT command line by the `HFILE` parameter.
   - A header file that contains the prototype of the function (MAPFUNC) that registers the mappings.
   - A C++ source file (with the extension `.cpp`) that contains the static methods called by OCCI while instantiating the objects; the filename is specified on the OTT command line by the `CPPFILE` parameter. Do not call these methods directly from your OCCI application.
   - A file that contains the function used to register the mappings with the environment (with the extension `.cpp`); the filename is specified on the OTT command line by the `MAPFILE` parameter.
   - A file (`OUTTYPE` file) that contains an entry for each of the translated types, including the version string and the file into which it is written; the filename is specified on the OTT command line by the `OUTTYPE` parameter.

4. Write the OCCI application and include the header files created by the OTT utility in the OCCI source code file.

   The application declares an environment and calls the function MAPFUNC to register the mappings.

5. Compile the OCCI application to create the OCCI object code, and link the object code with the OCCI libraries to create the program executable.

### C++ Classes Generated by the OTT Utility

When the OTT utility generates a C++ class from a database object type, the class declaration contains one element corresponding to each attribute of the object type. The data types of the attribute are mapped to types that are used in Oracle object data types, as defined in Table 8–2 on page 8-19.

For each class, two new operators, `readSQL()` and `writeSQL()` methods are generated. They are used by OCCI to marshall and unmarshall objects.

By default, the C++ classes generated by the OTT utility for an object type are derived from the `PObject` class, so the generated constructor in the class also derives from the `PObject` class. For inherited database types, the class is derived from the parent type class as is the generated constructor and only the elements corresponding to attributes not in the parent class are included.

Class declarations that include the elements corresponding to the database type attributes and the method declarations are included in the header file generated by the OTT utility. The method implementations are included in the `CPPFILE` file generated by the OTT utility.

**Example 8–14  How to Generate C++ Classes Using the OTT Utility**

This example demonstrates how to generate C++ classes using the OTT utility:

1. Define the types:
The OTT Utility and OCCI Applications

CREATE TYPE FULL_NAME AS OBJECT (first_name CHAR(20),
    last_name CHAR(20));
CREATE TYPE ADDRESS AS OBJECT (state CHAR(20), zip CHAR(20));
CREATE TYPE ADDRESS_TAB AS VARRAY(3) of REF ADDRESS;
CREATE TYPE PERSON AS OBJECT (id NUMBER, name FULL_NAME,
    curr_addr REF ADDRESS, prev_addr_1 ADDRESS_TAB) NOT FINAL;
CREATE TYPE STUDENT UNDER PERSON (school_name CHAR(20));

2. Provide an INTYPE file:

 CASE = SAME
 MAPFILE = RegisterMappings_3.cpp
 TYPE FULL_NAME AS FullName
    TRANSLATE first_name as FirstName
        last_name as LastName
 TYPE ADDRESS
 TYPE PERSON
 TYPE STUDENT

3. Invoke the OTT utility:

 ott userid=demousr intype=demoin_3.typ outype=demoout_3.typ
    code=cpp hfile=demo_3.h cppfile=demo_3.cpp

Map Registry Function

One function to register the mappings with the environment is generated by the OTT utility. The function contains the mappings for all the types translated by the invocation of the OTT utility. The function name is either specified in the MAPFUNC parameter or, if that parameter is not specified, derived from MAPFILE parameter. The only argument to the function is the pointer to Environment.

The function uses the provided Environment to get Map and then registers the mapping of each translated type.

Extending C++ Classes

To enhance the functionality of a class generated by the OTT utility, you can derive new classes. You can also add methods to a class, but Oracle does not recommend doing so due to an inherent risk.

See Also: "Carrying Forward User Added Code" on page 8-27 for details on how to use OTT markers to retain code you want to add in OTT generated files

To generate both CAddress and MyAddress classes from the SQL object type ADDRESS, MyAddress class can be derived from CAddress class. The OTT utility must then alter the code it generates in the following ways:

- By using the MyAddress class instead of the CAddress class to represent attributes whose database type is ADDRESS
- By using the MyAddress class instead of the CAddress class to represent vector and REF elements whose database type is ADDRESS
- By using the MyAddress class instead of the CAddress class as the base class for database object types that are inherited from ADDRESS. Even though a derived class is a subtype of MyAddress, the readSQL() and writeSQL() methods called are those of the CAddress class.
Example 8–15  How to Extend C++ Classes Using the OTT Utility

To use the OTT utility to generate the CAddress class, which is derived from MyAddress class), the following clause must be specified in the TYPE statement:

```
TYPE ADDRESS GENERATE CAddress AS MyAddress
```

Given the database types FULL_NAME, ADDRESS, PERSON, and PFGRFDTENT as they were created before and changing the INTYPE file to include the GENERATE...AS clause:

```
CASE = SAME
MAPFILE = RegisterMappings_5.cpp

TYPE FULL_NAME GENERATE CFullName AS MyFullName
   TRANSLATE first_name as FirstName
      last_name as LastName

TYPE ADDRESS GENERATE CAddress AS MyAddress
TYPE PERSON GENERATE CPerson AS MyPerson
TYPE STUDENT GENERATE CStudent AS MyStudent
```

Carrying Forward User Added Code

To extend the functionality of OTT generated code, at times programmers may want to add code in the OTT generated file. The way OTT can distinguish between OTT generated code and code added by the user is by looking for some predefined markers (tags). OTT recognizes OTT_USERCODE_START as the start of user code marker, and OTT_USERCODE_END as the end of user code marker.

For OTT marker support, a user block is defined as

```
OTT_USERCODE_START + user added code + OTT_USERCODE_END
```

OTT marker support enables carrying forward the user added blocks in *.h and *.cpp files.

Properties of OTT Markers

These items describe the properties of OTT Markers Support:

1. User must use the command line option USE_MARKER=TRUE from the very first time OTT is invoked to generate a file.
2. User should treat markers like other C++ statements; a marker defined by OTT in the generated file as follows when the command line option USE_MARKER=TRUE is used:

```
#ifndef OTT_USERCODE_START
#define OTT_USERCODE_START
#endif
#ifndef OTT_USERCODE_END
#define OTT_USERCODE_END
#endif
```
3. The markers, `OTT_USERCODE_START` and `OTT_USERCODE_END`, must be preceded and followed by white space.

4. OTT copies the text or code given within markers verbatim, along with the markers, while generating the code next time.

User modified code:

```
1  // --- modified generated code
2  OTT_USERCODE_START
3  // --- including "myfullname.h"
4  #ifndef MYFULLNAME_ORACLE
5  #include "myfullname.h"
6  #endif
7  OTT_USERCODE_END
8  // --- end of code addition
```

Carried forward code:

```
1  OTT_USERCODE_START
2  // --- including "myfullname.h"
3  #ifndef MYFULLNAME_ORACLE
4  #include "myfullname.h"
5  #endif
6  OTT_USERCODE_END
```

5. OTT does not carry forward user-added code properly if the database `TYPE` or `INTYPE` file undergoes changes as shown in the following cases:

   - If user modifies the case of the type name, OTT fails to determine the class name with which the code was associated earlier, as the case of the class name is modified by the user in the `INTYPE` file.

```
CASE=UPPER      CASE=LOWER
TYPE employee   TYPE employee
TRANSLATE SALARY$ AS salary TRANSLATE SALARY$ AS salary
DEPTNO AS department DEPTNO AS department
```

   - If user asks to generate the class with a different name (`GENERATE AS` clause of `INTYPE` file), OTT fails to determine the class name with which the code was associated earlier as the class name was modified by the user in the `INTYPE` file.

```
CASE=UPPER      CASE=LOWER
TYPE employee   TYPE employee
TRANSLATE SALARY$ AS salary TRANSLATE SALARY$ AS salary
DEPTNO AS department DEPTNO AS department
```

6. If OTT encounters an error while parsing an `.h` or `.cpp` file, it reports the error and leaves the file having error as it is so that the user can go back and correct the error reported, and rerun OTT.

7. OTT flags an error if:
■ it does not find a matching OTT_USERCODE_END for OTT_USERCODE_START encountered

■ markers are nested (OTT finds next OTT_USERCODE_START before OTT_USERCODE_END is found for the previous OTT_USERCODE_START)

■ OTT_USERCODE_END is encountered before OTT_USERCODE_START

Using OTT Markers

The user must use command line option USE_MARKER=TRUE to turn on marker support. There are two general ways in which OTT markers can carry forward user added code:

1. User code added in .h file.
   ■ User code added in global scope. This is typically the case when user must include different header files, forward declaration, and so on. Refer to the code example provided later.
   ■ User code added in class declaration. At any point of time OTT generated class declaration has private scope for data members and public scope for methods, or protected scope for data members and public scope for methods. User blocks can be added after all OTT generated declarations in either access specifiers.

Example 8–16 How to Add User Code to a Header File Using OTT Utility

```cpp
... #ifndef OTT_USERCODE_START
#define OTT_USERCODE_START
#endif
#ifndef OTT_USERCODE_END
#define OTT_USERCODE_END
#endif

#ifndef OCCI_ORACLE
#include <occi.h>
#endif

OTT_USERCODE_START // user added code ...
OTT_USERCODE_END

#ifndef ...
#include " ... "
#endif

OTT_USERCODE_START // user added code ...
OTT_USERCODE_END

class <class_name_1> : public oracle::occi::PObject
{
protected:
    ...
    // OTT generated data members
    OTT_USERCODE_START // user added code for data member / method
    ...
    // declaration / inline method
    OTT_USERCODE_END

public:
    void *operator new(size_t size);
```
User code added in .cpp file. OTT supports adding a new user defined method within OTT markers. The user block must be added at the beginning of the file, just after the includes and before the definition of OTT-generated methods. If there are multiple OTT-generated includes, user code can also be added between OTT generated includes. User code added in any other part of a xxx.cpp file is not carried forward.

Example 8–17 How to Add User Code to the Source File Using the OTT Utility

```cpp
#ifndef OTT_USERCODE_START
#define OTT_USERCODE_START
#endif

#ifndef OTT_USERCODE_END
#define OTT_USERCODE_END
#endif

# ifndef OTT_USERCODE_START
# define OTT_USERCODE_START
# endif

# ifndef OTT_USERCODE_END
# define OTT_USERCODE_END
# endif

*........................*
// generated method implementations for the ... object type.
*........................*

void *<class_name_1>::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

*........................*
// end of .cpp file
```
This chapter describes OCCI support for multibyte and Unicode charactersets.

This chapter contains these topics:

- Overview of Globalization and Unicode Support
- Specifying Charactersets
- Data Types for Globalization and Unicode Support
- Objects and OTT Support

Overview of Globalization and Unicode Support

OCCI now enables application development in all Oracle supported multibyte and Unicode charactersets. The UTF16 encoding of Unicode is fully supported. Application programs can specify their charactersets when the OCCI Environment is created. OCCI interfaces that take character string arguments (such as SQL statements, usernames, error messages, object names, and so on) have been extended to handle data in any characterset. Character data from relational tables or objects can be in any characterset. OCCI can be used to develop multi-lingual, global and Unicode applications.

Specifying Charactersets

OCCI applications must specify the client characterset and client national characterset when initializing the OCCI Environment. The client characterset specifies the characterset for all SQL statements, object/user names, error messages, and data of all CHAR data type (CHAR, VARCHAR2, LONG) columns/attributes. The client national characterset specifies the characterset for data of all NCHAR data type (NCHAR, NVARCHAR2) columns/attributes.

A new createEnvironment() interface that takes the client characterset and client national characterset is now provided. This allows OCCI applications to set characterset information dynamically, independent of the NLS_LANG and NLS_CHAR initialization parameter.

Note that if an application specifies OCCIUTF16 as the client characterset (first argument), then the application should use only the UTF16 interfaces of OCCI. These interfaces take UString argument types.

The charactersets in the OCCI Environment are client-side only. They indicate the charactersets the OCCI application uses to interact with Oracle. The database characterset and database national characterset are specified when the database is
created. Oracle converts all data from the client characterset/national characterset to the database characterset/national characterset before the server processes the data.

Example 9–1  How to Use Globalization and Unicode Support

Environment *env = Environment::createEnvironment("JA16SJIS","UTF8");

This statement creates an OCCI Environment with JA16SJIS as the client characterset and UTF8 as the client national characterset.

Any valid Oracle characterset name (except AL16UTF16) can be passed to createEnvironment(). An OCCI specific string OCCIUTF16 (in uppercase) can be passed to specify UTF16 as the characterset.

Environment *env = Environment::createEnvironment("OCCIUTF16","OCCIUTF16");
Environment *env = Environment::createEnvironment("US7ASCII", "OCCIUTF16");

Data Types for Globalization and Unicode Support

The data types used for supporting globalization and use of unicode include UString Data Type, Multibyte and UTF16 data, and CLOB and NCLOB Data Types.

UString Data Type

UString is a data type that enables applications and the OCMI library to pass and receive Unicode data in UTF-16 encoding. UString is templated from the C++ STL basic_string with Oracle’s utext data type.

typedef basic_string<utext> UString;

Oracle’s utext data type is a 2 byte short data type and represents Unicode characters in the UTF-16 encoding. A Unicode character’s codepoint can be represented in 1 utext or 2 utexts (2 or 4 bytes). Characters from European and most Asian scripts are represented in a single utext. Supplementary characters defined in the Unicode 3.1 standard are represented with 2 utext elements.

In Microsoft Windows platforms, UString is equivalent to the C++ standard wstring data type. This is because the wchar_t data type is type defined to a 2 byte short in these platforms, which is same as Oracle’s utext, allowing applications to use a wstring type variable where a UString would be normally required. Consequently, applications can also pass wide-character string literals, created by prefixing the literal with the letter ‘L’, to OCCI Unicode APIs.

Example 9–2  Using wstring Data Type

//bind Unicode data using wstring data type
//binding the Euro symbol, UTF16 codepoint 0x20AC
wchar_t eurochars[] = {0x20AC,0x00};
wstring eurostr(eurochars);
stmt->setUString(1,eurostr);

//Call the Unicode version of createConnection by
//passing widechar literals
Connection *conn = Connection(L"HR",L"password",L"");

OCCI applications should use the UString data type for data in UTF16 characterset
Data Types for Globalization and Unicode Support

Multibyte and UTF16 data

For data in multibyte character sets like JA16SJIS and UTF8, applications should use the C++ string type. The existing OCCI APIs that take string arguments can handle data in any multibyte character set. Due to the use of string type, OCCI supports only byte length semantics for multibyte character set strings.

Example 9–3 Binding UTF8 Data Using the string Data Type

```c++
//bind UTF8 data
//binding the Euro symbol, UTF8 codepoint : 0xE282AC
char eurochars[] = {0xE2,0x82,0xAC,0x00};
string eurostr(eurochars)
stmt->setString(1,eurostr);//use the string interface
```

For Unicode data in the UTF16 character set, the OCCI specific data type: UString and the OCCI UTF16 interfaces must be used.

Example 9–4 Binding UTF16 Data Using the UString Data Type

```c++
//bind Unicode data using UString data type
//binding the Euro symbol, UTF16 codepoint 0x20AC
utext eurochars[] = {0x20AC,0x00};
UString eurostr(eurochars);
stmt->setUString(1,eurostr);//use the UString interface
```

CLOB and NCLOB Data Types

Oracle provides the CLOB and NCLOB data types for storing and processing large amounts of character data. CLOBs represent data in the database character set and NCLOBs represent data in the database national character set. CLOBs and NCLOBs can be used as column types in relational tables and as attributes in object types.

The OCCI Clob class is used to work with both CLOB and NCLOB data types. If the database type is NCLOB, then the Clob set CharSetForm() method should be called with OCCI_SQLCS_NCHAR before reading/writing from the LOB.

The OCCI Clob class has support for multibyte and UTF16 character sets. By default, the Clob interfaces assume the data is encoded in the client-side character set (for both CLOBs and NCLOBs). To specify a different character set or to specify the client-side national character set for a NCLOB, call the set CharSetId() or set CharSetIdUString() methods with the appropriate character set. The OCCI specific string 'OCCIUTF16' can be passed to indicate UTF16 as the character set.

Example 9–5 Using CLOB and NCLOB Data Types

```c++
//client character set – ZHT16BIG5, national character set – UTF16
Environment *env = Environment::createEnvironment("ZHT16BIG5","OCCIUTF16");
...
Clob nclobvar;
//for NCLOBs, must call set CharSetForm method.
nclobvar.set CharSetForm(OCCI_SQLCS_NCHAR);
...
//if reading/writing data in UTF16 for this NCLOB, still must
//explicitly call set CharSetId
nclobvar.set CharSetId("OCCIUTF16")
```

To read or write data in multibyte character sets, use the existing read and write interfaces that take a char buffer. New overloaded interfaces that take utext buffers for
UTF16 data have been added to the Clob Class as read(), write() and writeChunk() methods. The arguments and return values for these methods are either bytes or characters, depending on the characterset of the LOB.

Objects and OTT Support

Multibyte and UTF16 character sets are supported for handling character data in object attributes. All CHAR data type (CHAR or VARCHAR2) attributes hold data in the client-side characterset, while all NCHAR data type (NCHAR or NVARCHAR2) attributes hold data in the client-side national characterset. A member variable of UString data type represents an attribute in UTF16 characterset.

See Also:

- Chapter 13, "OCCI Application Programming Interface": two new versions of operator new() on page 13-165 that have been added to the POBJ ect Class for object support
- Chapter 8, "Object Type Translator Utility": a new UNICODE parameter on page 8-9 that has been added for OTT utility support.
This chapter describes the OCCI implementation of Oracle Streams Advanced Queuing (AQ) for messages.

This chapter contains these topics:

- Overview of Oracle Streams Advanced Queuing
- AQ Implementation in OCCI
- Creating Messages
- Enqueuing Messages
- Dequeuing Messages
- Listening for Messages
- Registering for Notification
- Message Format Transformation

See Also:

- Oracle Streams Advanced Queuing User's Guide for basic concepts of Advanced Queuing
- Chapter 13, "OCCI Application Programming Interface"

Overview of Oracle Streams Advanced Queuing

Oracle Streams is a new information sharing feature that provides replication, message queuing, data warehouse loading, and event notification. It is also the foundation behind Oracle Streams Advanced Queuing (AQ).

Advanced Queuing is the integrated message queuing feature that exposes message queuing capabilities of Oracle Streams. AQ enables applications to:

- Perform message queuing operations similar to SQL operations from the Oracle database
- Communicate asynchronously through messages in AQ queues
- Integrate with database for unprecedented levels of operational simplicity, reliability, and security to message queuing
- Audit and track messages
- Supports both synchronous and asynchronous modes of communication
The advantages of using AQ in OCCI applications include:

- Create applications that communicate with each other in a consistent, reliable, secure, and autonomous manner
- Store messages in database tables, bringing the reliability and recoverability of the database to your messaging infrastructure
- Retain messages in the database automatically for auditing and business intelligence
- Create applications that leverage messaging without having to deal with a different security, data type, or operational mode
- Leverage transactional characteristics of the database

Since traditional messaging solutions have single subscriber queues, a queue must be created for each pair of applications that communicate with each other. The publish/subscribe protocol of the AQ makes it easy to add additional applications (subscribers) to a conversation between multiple applications.

**AQ Implementation in OCCI**

OCCI AQ is a set of interfaces that allows messaging clients to access the Advanced Queuing feature of Oracle for enterprise messaging applications. Currently, OCCI AQ supports only the operational interfaces and not the administrative interface, but administrative operations can be accessed through embedded PL/SQL calls.

**See Also:** Package DBMS_AQADM in Oracle Database PL/SQL Packages and Types Reference for administrative operations in AQ support through PL/SQL

The AQ feature can be used with other interfaces available through OCCI for sending, receiving, publishing, and subscribing in a message-enabled database. Synchronous and asynchronous message consumption is available based on a message selection rule.

Enqueuing refers to sending a message to a queue and dequeuing refers to receiving one. A client application can create a message, set the desired properties on it and enqueue it by storing the message in the queue, a table in the database. When dequeuing a message, an application can either dequeue it synchronously by calling receive methods on the queue, or asynchronously by waiting for a notification from the database.

The AQ feature is implemented through the abstractions **Message, Agent, Producer, Consumer, Listener** and Subscription.

**Message**

A message is the basic unit of information being inserted into and retrieved from a queue. A message consists of control information and payload data. The control information represents message properties used by AQ to manage messages. The payload data is the information stored in the queue and is transparent to AQ.
Agent

An Agent represents and identifies a user of the queue, either producer or consumer of the message, either an end-user or an application. An Agent is identified by a name, an address and a protocol. The name can be either assigned by the application, or be the application itself. The address is determined in terms of the communication protocol. If the protocol is 0 (default), the address is of the form [schema.]queuename[@dblink], a database link.

Agents on the same queue must have a unique combination of name, address, and protocol. Example 10–1 demonstrates an instantiation of a new Agent object in a client program.

Example 10–1 Creating an Agent

Agent agt(env, "Billing_app", "billqueue", 0);

Producer

A client uses a Producer object to enqueue Messages into a queue. It is also used to specify various enqueue options.

See Also: Producer Class documentation in Chapter 13, "OCCI Application Programming Interface"

Consumer

A client uses a Consumer object to dequeue Messages that have been delivered to a queue. It also specifies various dequeuing options.

Before a consumer can receive messages,

Example 10–2 Setting the Agent on the Consumer

Consumer cons(conn);
...
cons.setAgent(ag);
cons.receive();

See Also: Consumer Class documentation in Chapter 13, "OCCI Application Programming Interface"

Listener

A Listener listens for Messages for registered Agents at specified queues.

See Also: Listener Class documentation in Chapter 13, "OCCI Application Programming Interface"
Subscription

A Subscription encapsulates the information and operations necessary for registering a subscriber for notifications.

Creating Messages

As mentioned previously, a Message is a basic unit of information that contains both the properties of the message and its content, or payload. Each message is enqueued by the Producer and dequeued by the Consumer objects.

Message Payloads

OCCI supports three types of message payloads: RAW, AnyData, and User-defined.

RAW

RAW payloads are mapped as objects of the Bytes Class in OCCI.

AnyData

The AnyData type models self-descriptive data encapsulation; it contains both the type information and the actual data value. Data values of most SQL types can be converted to AnyData, and then be converted to the original data type. AnyData also supports user-defined data types. The advantage of using AnyData payloads is that it ensures both type preservation after an enqueue and dequeue process, and that it allows the user to use a single queue for all types used in the application. Example 10–3 demonstrates how to create an AnyData message. Example 10–4 shows how to retrieve the original data type from the message.

**Example 10–3 Creating an AnyData Message with a String Payload**

```cpp
AnyData any(conn);
any.setFromString("item1");
Message mes(env);
mes.setAnyData(any);
```

**Example 10–4 Determining the Type of the Payload in an AnyData Message**

```cpp
TypeCode tc = any.getType();
```

User-defined

OCCI supports enqueuing and dequeuing of user-defined types as payloads. Example 10–5 demonstrates how to create a payload with a user-defined Employee object.

**Example 10–5 Creating an User-defined Payload**

```cpp
// Assuming type Employee { name varchar2(25),
//                         deptid number(10),
//                         manager varchar2(25) }
Employee *emp = new Employee();
emp.setName("Scott");
emp.setDeptid(10);
emp.setManager("James");
Message mes(env);
mes.setObject(emp);
```
Message Properties

Aside from payloads, the user can specify several additional message properties, such as Correlation, Sender, Delay and Expiration, Recipients, and Priority and Ordering.

Correlation
Applications can specify a correlation identifier of the message during the enqueuing process, as demonstrated in Example 10–6. This identifier can then be used by the dequeuing application.

Example 10–6  Specifying the Correlation identifier
mes.setCorrelationId("enq_corr_di");

Sender
Applications can specify the sender of the message, as demonstrated in Example 10–7. The sender identifier can then be used by the receiver of the message.

Example 10–7  Specifying the Sender identifier
mes.setSenderId(agt);

Delay and Expiration
Time settings control the delay and expiration times of the message in seconds, as demonstrated in Example 10–8.

Example 10–8  Specifying the Delay and Expiration times of the message
mes.setDelay(10);
mes.setExpirationTime(60);

Recipients
The agents for whom the message is intended can be specified during message encoding, as demonstrated in Example 10–9. This ensures that only the specified recipients can access the message.

Example 10–9  Specifying message recipients
vector<Agent> agt_list;
for (i=0; i<num_recipients; i++)
    agt_list.push_back(Agent(name, address, protocol));
mes.setRecipientList(agt_list);

Priority and Ordering
By assigning a priority level to a message, the sender can control the order in which the messages are dequeued by the receiver. Example 10–10 demonstrates how to set the priority of a message.

Example 10–10  Specifying the Priority of a Message
mes.setPriority(3);
Enqueuing Messages

Messages are enqueued by the Producer. The Producer Class is also used to specify enqueue options. A Producer object can be created on a valid connection where enqueuing is performed, as illustrated in Example 10–11.

The transactional behavior of the enqueue operation can be defined based on application requirements. The application can make the effect of the enqueue operation visible externally either immediately after it is completed, as in Example 10–11, or only after the enclosing transaction has been committed.

To enqueue the message, use the send() method, as demonstrated in Example 10–11. A client may retain the Message object after it is sent, modify it, and send it again.

Example 10–11 Creating a Producer, Setting Visibility, and Enqueuing the Message

Producer prod(conn);
...
prod.setVisibility(Producer::ENQ_IMMEDIATE);
...
Message mes(env);
...
mes.setBytes(obj); // obj represents the content of the message
prod.send(mes, queueName); // queueName is the name of the queue

Dequeuing Messages

Messages delivered to a queue are dequeued by the Consumer. The Consumer Class is also used to specify dequeue options. A Consumer object can be created on a valid connection to the database where a queue exists, as demonstrated in Example 10–12.

In applications that support multiple consumers in the same queue, the name of the consumer has to be specified as a registered subscriber to the queue, as shown in Example 10–12.

To dequeue the message, use the receive() method, as demonstrated in Example 10–12. The typeName and schemaName parameters of the receive() method specify the type of payload and the schema of the payload type.

Example 10–12 Creating a Consumer, Naming the Consumer, and Receiving a Message

Consumer cons(conn);
...
// Name must be registered with the queue through administrative interface
cons.setConsumerName("BillApp");
cons.setQueueName(queueName);
...
Message mes = cons.receive(Message::OBJECT, "BILL_TYPE", "BILL_PROCESSOR");
...
// obj is assigned the content of the message
obj = mes.getObject();

When the queue payload type is either RAW or AnyData, schemaName and typeName are optional, but you must specify these parameters explicitly when working with user-defined payloads. This is illustrated in Example 10–13.

Example 10–13 Receiving a Message

//receiving a RAW message
Dequeueing Options

The dequeuing application can specify several dequeuing options before it begins to receive messages. These include **Correlation**, **Mode**, and **Navigation**.

**Correlation**

The message can be dequeued based on the value of its correlation identifier using the `setCorrelationId()` method, as shown in Example 10–14.

**Mode**

Based on application requirements, the user can choose to only browse through messages in the queue, remove the messages from the queue, or lock messages using the `setDequeueMode()` method, as shown in Example 10–14.

**Navigation**

Messages enqueued in a single transaction can be viewed as a single group by implementing the `setPositionOfMessage()` method, as shown in Example 10–14.

Example 10–14  Specifying dequeuing options

```java
cons.setCorrelationId(corrId);
...
cons.setDequeueMode(deqMode);
...
cons.setPositionOfMessage(Consumer::DEQ_NEXT_TRANSACTION);
```

Listening for Messages

The Listener listens for messages on queues on behalf of its registered clients. The **Listener Class** implements the `listen()` method, which is a blocking call that returns when a queue has a message for a registered agent, or throws an error when the time out period expires. Example 10–15 illustrates the listening protocol.

Example 10–15  Listening for messages

```java
Listener listener(conn);

vector<Agent> agtList;
for(int i=0; i<num_agents; i++)
    agtList.push_back(Agent(name, address, protocol);

listener.setAgentList(agtList);
listener.setTimeOutForListen(10);

Agent agt(env);
try{
    agt = listener.listen();
}
catch{
    cout<<e.getMessage()<<endl;
}

Registering for Notification

The Subscription Class implements the publish-subscribe notification feature. It allows an OCCI AQ application to receive client notifications directly, register an e-mail address to which notifications can be sent, register an HTTP URL to which notifications can be posted, or register a PL/SQL procedure to be invoked on a notification. Registered clients are notified asynchronously when events are triggered or on an explicit AQ enqueue. Clients do not have to be connected to a database.

An OCCI application can do all of the following:

- Register interest in notification in the AQ namespace, and be notified when an enqueue occurs.
- Register interest in subscriptions to database events, and receive notifications when these events are triggered.
- Manage registrations, such as disable registrations temporarily, or dropping registrations entirely.
- Post (or send) notifications to registered clients.

Publish-Subscribe Notifications

Notifications can work in several ways. They can be:

- received directly by the OCCI application
- sent to a pre-specified e-mail address
- sent to a pre-defined HTTP URL
- invoke a pre-specified database PL/SQL procedure

Registered clients are notified asynchronously when events are triggered, or on an explicit AQ enqueue. Clients do not have to be connected to a database for notifications to work. Registration can be accomplished either as Direct Registration or Open Registration.

Direct Registration

You can register directly with the database. This is relatively simple, and the registration takes effect immediately. Example 10–16 outlines the required steps to successfully register for direct event notification. It is assumed that the appropriate event trigger or queue is in existence, and that the initialization parameter COMPATIBLE is set to 8.1 or higher.

Example 10–16   How to Register for Notifications; Direct Registration

1. Create the environment in Environment::EVENTS mode.
2. Create the Subscription object.
3. Set these Subscription attributes.
   The namespace can be set to these options:
   - To receive notifications from AQ queues, namespace must be set to Subscription::NS_AQ. The subscription name is then either of the form
SCHEMA.QUEUE when registering on a single consumer queue, or
SCHEMA.QUEUE:CONSUMER_NAME when registering on a multiconsumer
queue.

- To receive notifications from other applications that use
  conn->postToSubscription() method, namespace must be set to
  Subscription::NS_ANONYMOUS.

The protocol can be set to these options:

- If an OCCI client must receive an event notification, this attribute should be
  set to Subscription::PROTO_CBK. You also must set the notification
  callback and the subscription context before registering the Subscription.
  The notification callback is called when the event occurs.

- For an e-mail notification, set the protocol to Subscription::PROTO_MAIL.
  You must set the recipient name before subscribing to avoid an application
  error.

- For an HTTP URL notification, set the protocol to Subscription::PROTO_HTTP.
  You must set the recipient name before subscribing to avoid an application
  error.

- To invoke PL/SQL procedures in the database on event notification, set
  protocol to Subscription::PROTO_SERVER. You must set the recipient
  name before subscribing to avoid an application error.

4. Register the subscriptions using connection->registerSubscriptions().

Open Registration

You can also register through an intermediate LDAP that sends the registration request
to the database. This is used when the client cannot have a direct database connection;
for example, the client wants to register for an open event while the database is down.
This approach is also used when a client wants to register for the same event(s) in
multiple databases, concurrently.

Example 10–17 outlines the LDAP open registration using the Oracle Enterprise
Security Manager (OESM). Open registration has these prerequisites:

- The client must be an enterprise user
  - In each enterprise domain, create an enterprise role
    ENTERPRISE_AQ_USER_ROLE
  - For each database in the enterprise domain, add a global role
    GLOBAL_AQ_USER_ROLE to enterprise the role
    ENTERPRISE_AQ_USER_ROLE.
  - For each enterprise domain, add an enterprise role
    ENTERPRISE_AQ_USER_ROLE to the privilege group cn=OracleDBAQUsers under
cn=oraclecontext in the administrative context
  - For each enterprise user that is authorized to register for events in the
database, grant enterprise the role ENTERPRISE_AQ_USER_ROLE.

- The compatibility of the database must be 9.0 or higher

- LDAP_REGISTRATION_ENABLED must be set to TRUE (default is FALSE):
  ALTER SYSTEM SET LDAP_REGISTRATION_ENABLED=TRUE

- LDAP_REG_SYNC_INTERVAL must be set to the time_interval (in seconds) to
  refresh registrations from LDAP (default is 0, do not refresh):
Registering for Notification

ALTER SYSTEM SET LDAP_REG_SYNC_INTERVAL = time_interval

To force a database refresh of LDAP registration information immediately, issue this command:

ALTER SYSTEM REFRESH LDAP_REGISTRATION

**Example 10–17  How to Use Open Registration with LDAP**

1. Create the environment in `Environment::EVENTS|Environment::USE_LDAP` mode.

2. Set the `Environment` object for accessing LDAP:
   - The host and port on which the LDAP server is residing and listening
   - The authentication method; only simple username and password authentication is currently supported
   - The username (distinguished name) and password for authentication with the LDAP server
   - The administrative context for Oracle in the LDAP server

3. Create the `Subscription` object.

4. Set the distinguished names of the databases in which the client wants to receive notifications on the `Subscription` object.

5. Set these `Subscription` attributes.

   The `namespace` can be set to these options:
   - To receive notifications from AQ queues, `namespace` must be set to `Subscription::NS_AQ`. The subscription name is then either of the form `SCHEMA.QUEUE` when registering on a single consumer queue, or `SCHEMA.QUEUE:CONSUMER_NAME` when registering on a multiconsumer queue.
   - To receive notifications from other applications that use `conn->postToSubscription()` method, `namespace` must be set to `Subscription::NS_ANONYMOUS`

   The `protocol` can be set to these options:
   - If an OCCI client must receive an event notification, this attribute should be set to `Subscription::PROTO_CBK`. You also must set the notification callback and the subscription context before registering the `Subscription`. The notification callback is called when the event occurs.
   - For an e-mail notification, set the protocol to `Subscription::PROTO_MAIL`. You must then set the recipient name to the e-mail address to which the notifications must be sent.
   - For an HTTP URL notification, set the protocol to `Subscription::HTTP`. You must set the recipient name to the URL to which the notification must be posted.
   - To invoke PL/SQL procedures in the database on event notification, set protocol to `Subscription::PROTO_SERVER`. You must set the recipient name to the database procedure invoked on notification.

6. Register the subscription: `environment->registerSubscriptions()`.
Open registration takes effect when the database accesses LDAP to pick up new registrations. The frequency of pick-ups is determined by the value of REG_SYNC_INTERVAL.

Clients can temporarily disable subscriptions, re-enable them, or permanently unregister from future notifications.

Notification Callback

The client must register a notification callback. This callback is invoked only when there is some activity on the registered subscription. In the Streams AQ namespace, this happens when a message of interest is enqueued.

The callback must return 0, and it must have the following specification:

```c
typedef unsigned int (*callbackfn) (Subscription &sub, NotifyResult *nr);
```

where:

- The `sub` parameter is the Subscription object which was used when the callback was registered.
- The `nr` parameter is the NotifyResult object holding the notification info.

Ensure that the subscription object used to register for notifications is not destroyed until it explicitly unregisters the subscription.

The user can retrieve the payload, message, message id, queue name and consumer name from the NotifyResult object, depending on the source of notification. These results are summarized in Table 10–1. Only a bytes payload is currently supported, and you must explicitly dequeue messages from persistent queues in the AQ namespace. If notifications come from non-persistent queues, messages are available to the callback directly; only RAW payloads are supported. If notifications come from persistent queues, the message has to be explicitly dequeued; all payload types are supported.

<table>
<thead>
<tr>
<th>Notification Result Attribute</th>
<th>ANONYMOUS Namespace</th>
<th>AQ Namespace, Persistent Queue</th>
<th>AQ Namespace, Non-Persistent Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>payload</td>
<td>valid</td>
<td>invalid</td>
<td>invalid</td>
</tr>
<tr>
<td>message</td>
<td>invalid</td>
<td>invalid</td>
<td>valid</td>
</tr>
<tr>
<td>messageID</td>
<td>invalid</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td>consumer name</td>
<td>invalid</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td>queue name</td>
<td>invalid</td>
<td>valid</td>
<td>valid</td>
</tr>
</tbody>
</table>

Message Format Transformation

Applications often use data in different formats, and this requires a type transformation. A transformation is implemented as a SQL function that takes the source data type as input and returns an object of the target data type.

Transformations can be applied when message are enqueued, dequeued, or when they are propagated to a remote subscriber.
See Also: The following chapters of the Oracle Streams Advanced Queuing User's Guide for information of format transformation:

- Oracle Streams AQ Administrative Interface
- Oracle Streams AQ Administrative Interface: Views
- Oracle Streams AQ Operational Interface: Basic Operations
The Oracle XA library is an external interface that allows transaction managers other than the Oracle server to coordinate global transactions. The XA library supports non-Oracle resource managers in distributed transactions. This is particularly useful in transactions between several databases and resources.

The implementation of the Oracle XA library conforms to the X/Open Distributed Transaction Processing (DTP) software architecture's XA interface specification. The Oracle XA Library is installed as part of the Oracle Database Enterprise Edition.

This chapter contains these topics:

- Application Development with XA and OCCI
- APIs for XA Support

See Also:

- [http://www.opengroup.org](http://www.opengroup.org)
- Oracle Database Advanced Application Developer’s Guide for more details on the Oracle XA library and architecture
- Chapter 13, "OCCI Application Programming Interface"

### Application Development with XA and OCCI

For connection, disconnection, and transaction control on Oracle databases, applications must interface with a transaction manager. OCCI has APIs for interacting with Environment and Connection objects within XA and make them available for Oracle database access, such as SELECT queries, DML statements, object access, and so on.

**Example 11–1  How to Use Transaction Managers with XA**

/* Transaction manager opens connection to the Oracle server*/
 tpopen("oracle_xa+acc=p/HR/password+sestm=10", 1, TMNFLAGS);
/* Transaction manager issues XA commands to start a global transaction*/
 tpbegin();

/* Access the underlying Oracle database using OCCI */
Environment *xaenv = Environment::getXAEnvironment(  
  "oracle_xa+acc=p/HR/password+sestm=10");
Connection *xaconn = xaenv->getXACConnection(  
  "oracle_xa+acc=p/HR/password+sestm=10");

/* Use the Environment & Connection objects */
Statement *stmt = xaconn->createStatement(
    "Update Emp set sal = sal * 0.2");

...

/* Release the Environment & Connection objects */
xaenv->releaseXAConnection(xaconn);
Environment::releaseXAEnvironment(xaenv);

### APIs for XA Support

The following methods of the Environment Class support use of XA libraries:

- `getXAConnection()` on page 13-92
- `releaseXAEnvironment()` on page 13-93
- `releaseXAConnection()` on page 13-92
- `releaseXAEnvironment()` on page 13-93

In addition, the `getXAErrorCode()` method of SQLException Class on page 13-202 is necessary for XA enabled applications to determine if thrown exceptions are due to an SQL error (`XA_OK`) or an XA error (an XA error code).
Optimizing Performance of OCCI Applications

This chapter describes a few suggestions that lead to better performance for your OCCI custom applications.

This chapter contains these topics:

- Transparent Application Failover
- Connection Sharing
- Application-Managed Data Buffering
- Array Fetch Using next() Method
- Modifying Rows Iteratively
- Run-time Load Balancing of the Stateless Connection Pool
- Fault Diagnosability
- Client Result Cache

See Also:
- Chapter 13, "OCCI Application Programming Interface"

Transparent Application Failover

OCCI Transparent Application Failover (TAF) enables OCCI to be more robust in handling database instance failures in distributed applications at run time. If a server node becomes unavailable, applications automatically reconnect to another surviving node.

TAF occurs when the client application, during a roundtrip operation, detects that the database instance is down. It establishes a connection to the backup database configured for TAF. This backup can be another node in an Oracle RAC configuration, a hot standby database, or the same database instance itself.

The OCCI/OCI API responsible for the roundtrip on the failed connection will typically return one of the following errors:

- ORA-25401: can not continue fetches
- ORA-25402: transaction must roll back
- ORA-25408: can not safely replay call

The new connection is may be used for subsequent application requests and for any ongoing work that must be restarted.
Idle connections in the application are not affected.

Some design options should be considered when including Transparent Application Failover in an application:

■ Because of the delays inherent to failover processing, the design of the application may include a notice to the user that a failover is in progress and that normal operation should resume shortly.

■ If the session on the initial instance received \texttt{ALTER SESSION} commands before the failover began, they are not automatically replayed on the second instance. Consequently, the developer may want to replay these \texttt{ALTER SESSION} commands on the second instance.

It is the user’s responsibility to track changes to the \texttt{SESSION} parameters.

To address these problems, the application can register a failover callback function. After a failover, the callback function is invoked at different times while reestablishing the user’s session.

■ The first call to the callback function occurs when Oracle first detects an instance connection loss. This callback is intended to allow the application to inform the user of an upcoming delay.

■ If failover is successful, a second call to the callback function occurs when the connection is reestablished and usable. At this time the client may want to replay \texttt{ALTER SESSION} commands and inform the user that failover has happened. Note that you must keep track of \texttt{SESSION} parameter changes and then replay them after the failover is complete.

If failover is unsuccessful, then the callback function is called to inform the application that failover cannot take place.

■ An initial attempt at failover may not always successful. The failover callback should return \texttt{FO_RETRY} to indicate that the failover should be attempted again.

\textbf{See Also:}

■ Definition of \texttt{FailOverType} and \texttt{FailOverEventType} in \texttt{Table 13–11, "Enumerated Values Used by Connection Class" in Chapter 13, "OCCI Application Programming Interface"}

■ \textit{Oracle Database Net Services Administrator’s Guide} for more detailed information about application failover.

\section*{Using Transparent Application Failover}

To enable TAF, the connect string has to be configured for failover and registered on \texttt{Connection} (created from \texttt{Environment, ConnectionPool} and \texttt{StatelessConnectionPool}). To register the callback function, use the \texttt{Connection Class} interface \texttt{setTAFNotify()} on page 13-59:

\begin{verbatim}
void Connection::setTAFNotify(
  int (*notifyFn)(
    Environment *env,
    Connection *conn,
    void *ctx,
    FailOverType foType,
    FailOverEventType foEvent),
  void *ctxTAF);
\end{verbatim}
Note that TAF support for ConnectionPools does not include BACKUP and PRECONNECT clauses; these should not be used in the connect string.

**Objects and Transparent Application Failover**

Transparent application failover works with the OCCI navigational and associative access models and the object cache. In a non-Oracle RAC setup, you must ensure that the object type definitions and object OIDs in primary and backup instances are identical.

If the application receives ORA-25402: transaction must roll back error after the failover, then it must initiate a rollback to correctly reset the object cache on the client. If a transaction has not started before the failover, the application should still initiate a rollback after the failover to refresh the objects on the client object cache from the new instance.

**Connection Pooling and Transparent Application Failover**

If the transparent application failover feature is activated, connections created in a connection pool are also failed over. The application failover callback must be specified for each connection obtained from the connection pool; these connections are failed over when used after the primary instance failure.

Note that connections in a custom connection pool must be explicitly cleaned and repaired. Consider an application that has 500 connections in a pool; 10 of the connections are busy (doing a round-trip) and 490 are free or idle. If the database instance fails, then TAF will work on 10 active connections, and client requests on these connections must be restarted. When each of the other 490 connections are picked up by the application, TAF is performed and OCCI returns one of error codes ORA-25401, ORA-24502, or ORA-25408, forcing a restart of the user request. The application can avoid these errors on the 490 idle connections by repairing or purging these connections by using the knowledge that TAF was previously activated on 10 connections in the connection pool.

To repair connections in OCCI, use the Connection Class interface `getServerVersion()` on page 13-54, a light-weight, data-neutral database call for starting TAF on connections to failed instances:

```
string getServerVersion() const;
```

In the OCCI TAF callback, applications may invoke `getServerVersion()` on idle connections in the custom pool, to start and complete failover for these connections. `Example 12–1` demonstrates how to use OCCI for TAF callbacks and for repairing bad connections. Note that the example does not show custom pool data structure or mutexing and concurrency control.

Note that TAF behavior is the same for standalone connections and connections in a custom connection pool.

**Example 12–1  How to Enable TAF for Connection Pooling**

```cpp
#include "occi.h"
#include "iostream"
#include "time.h"

using namespace std;
using namespace oracle::occi;

//Application custom pool of 3 connections
```
```cpp
Environment *env;
Connection *conn1,*conn2,*conn3;
bool conn1free,conn2free,conn3free;
bool repairing = false;

int taf_callback(Environment *env, Connection *conn, void *ctx,
      Connection::FailOverType foType, Connection::FailOverEventType foEvent)
{
    cout << "TAF callback for connection " << conn << endl;

    if(foEvent == Connection::FO_ERROR)
    {
        cout << "Retrying" << endl;
        return FO_RETRY;
    }

    if (foEvent == Connection::FO_END)
    {
        cout << "TAF complete for connection " << conn << endl;
        if (repairing == false)
        {
            repairing = true;
            cout << "repairing other idle connections" << endl;

            //ignore errors during TAF
            try
            {
                if (conn1free) conn1->getServerVersion();
            }
            catch (...)
            {
            }
            try
            {
                if (conn2free) conn2->getServerVersion();
            }
            catch (...)
            {
            }
            try
            {
                if (conn3free) conn3->getServerVersion();
            }
            catch (...)
            {
            }
            repairing = false;
        } //if
    } //if

    return 0; //continue failover
}

main()
{
    try
    {
        env = Environment::createEnvironment(Environment::THREADED_MUTEXED);
        //open 3 connections;
        conn1 = env->createConnection("hr","password","inst1_failback");
```

Connection Sharing

This section covers the following topics:

- Introduction to Thread Safety
- Implementing Thread Safety
- Serialization
Introduction to Thread Safety

Threads are lightweight processes that exist within a larger process. Threads each share the same code and data segments, but have their own program counters, system registers, and stack. Global and static variables are common to all threads, and a mutual exclusivity mechanism may be required to manage access to these variables from multiple threads within an application.

Once spawned, threads run asynchronously to one another. They can access common data elements and make OCCI calls in any order. Because of this shared access to data elements, a mechanism is required to maintain the integrity of data being accessed by multiple threads. The mechanism to manage data access takes the form of mutexes (mutual exclusivity locks), which ensure that no conflicts arise between multiple threads that are accessing shared resources within an application. In OCCI, mutexes are granted on an OCCI environment basis.

This thread safety feature of the Oracle database server and OCCI library enables developers to use OCCI in a multithreaded application with these added benefits:

- Multiple threads of execution can make OCCI calls with the same result as successive calls made by a single thread.
- When multiple threads make OCCI calls, there are no side effects between threads.
- Even if you do not write a multithreaded program, you do not pay any performance penalty for including thread-safe OCCI calls.
- Use of multiple threads can improve program performance. You can discern gains on multiprocessor systems where threads run concurrently on separate processors, and on single processor systems where overlap can occur between slower operations and faster operations.

In addition to client/server applications, where the client can be a multithreaded program, thread safety is typically used in three-tier or client/agent/server architectures. In this architecture, the client is concerned only with presentation services. The agent (or application server) processes the application logic for the client application. Typically, this relationship is a many-to-one relationship, with multiple clients sharing the same application server.

The server tier in the three-tier architecture is an Oracle database server. The applications server (agent) supports multithreading, with each thread serving a separate client application. In an Oracle environment, this middle-tier application server is an OCCI or precompiler program.

Implementing Thread Safety

To take advantage of thread safety by using OCCI, an application must be running in a thread-safe operating system. Then the application must inform OCCI that the application is running in multithreaded mode by specifying THREAD_MUTEXED or THREAD_UNMUTEXED for the mode parameter of the createEnvironment() method. For example, to turn on mutual exclusivity locking, issue the following statement:

```cpp
Environment *env = Environment::createEnvironment(
    Environment::THREAD_MUTEXED);
```
Note that once `createEnvironment` is called with `THREADED_MUTEXED` or `THREADED_UNMUTEXED`, all subsequent calls to the `createEnvironment` method must also be made with `THREADED_MUTEXED` or `THREADED_UNMUTEXED` modes.

If a multithreaded application is running in a thread-safe operating system, then the OCCI library manages mutexes for the application on a for each-OCCI-environment basis. However, you can override this feature and have your application maintain its own mutex scheme. This is done by specifying a mode value of `THREADED_UNMUTEXED` to the `createEnvironment()` method.

Applications that run on non-thread-safe platforms should not pass a value of `THREADED_MUTEXED` or `THREADED_UNMUTEXED` to the `createEnvironment()` method.

If an application is single threaded, regardless of whether the platform is thread safe, the application should pass a value of `Environment::DEFAULT` to the `createEnvironment()` method. This is also the default value for the mode parameter. Single threaded applications which run in `THREADED_MUTEXED` mode may incur performance degradation.

OCCI does not support nonblocking mode.

**Serialization**

As an application programmer, you have two basic options regarding concurrency in a multithreaded application:

- Automatic serialization, in which you use OTIS’s transparent mechanisms
- Application-provided serialization, in which you manage the contingencies involved in maintaining multiple threads

**Automatic Serialization**

In cases where there are multiple threads operating on objects (connections and connection pools) derived from an OCCI environment, you can elect to let OCCI serialize access to those objects. The first step is to pass a value of `THREADED_MUTEXED` to the `createEnvironment()` method. At this point, the OCCI library automatically acquires a mutex on thread-safe objects in the environment.

When the OCCI environment is created with `THREADED_MUTEXED` mode, then only the `Environment`, `Map`, `ConnectionPool`, `StatelessConnectionPool` and `Connection` objects are thread-safe. That is, if two threads make simultaneous calls on one of these objects, then OCCI serializes them internally. However, note that all other OCCI objects, such as `Statement`, `ResultSet`, `SQLException`, `Stream`, and so on, are not thread-safe as, applications should not operate on these objects simultaneously from multiple threads.

Note that the bulk of processing for an OCCI call happens on the server, so if two threads that use OCCI calls go to the same connection, then one of them could be blocked while the other finishes processing at the server.

**Application-Provided Serialization**

In cases where there are multiple threads operating on objects derived from an OCCI environment, you can chose to manage serialization. The first step is to pass a value of `THREADED_UNMUTEXED` for the `createEnvironment` mode. In this case the application must mutual exclusively lock OCCI calls made on objects derived from the same OCCI environment. This has the advantage that the mutex scheme can be optimized based on the application design to gain greater concurrency.
When an OCCI environment is created in this mode, OCCI recognizes that the application is running in a multithreaded application, but that OCCI need not acquire its internal mutexes. OCCI assumes that all calls to methods of objects derived from that OCCI environment are serialized by the application. You can achieve this two different ways:

- Each thread has its own environment. That is, the environment and all objects derived from it (connections, connection pools, statements, result sets, and so on) are not shared across threads. In this case your application need not apply any mutexes.

- If the application shares an OCCI environment or any object derived from the environment across threads, then it must serialize access to those objects (by using a mutex, and so on) such that only one thread is calling an OCCI method on any of those objects.

In both cases, no mutexes are acquired by OCCI. You must ensure that only one OCCI call is in process on any object derived from the OCCI environment at any given time when THREADED_UNMUTEXED is used.

OCCI is optimized to reuse objects as much as possible. Since each environment has its own heap, multiple environments result in increased consumption of memory. Having multiple environments may imply duplicating work regarding connections, connection pools, statements, and result set objects. This results in further memory consumption.

Having multiple connections to the server results in more resource consumption on both the server and the network. Having multiple environments normally entails more connections.

### Operating System Considerations

Some operating systems provide facilities for spawning processes that allow child processes to reuse states created by their parent process.

After a parent process spawns a child process, the child process must not use the database connection created by the parent. Because SQL*Net expects only one user process to use a particular connection to the database, attempts by the child process to use the same database connection as the parent may cause undesired connection interference, and result in intermittent ORA-03137 errors.

When the application requires multiple concurrent connections, Oracle recommends using threads on platforms that support threading. Oracle supports concurrent connections in both single-threaded and multi-threaded applications.

See "Introduction to Thread Safety" on page 12-6 and "Implementing Thread Safety" on page 12-6 for more information about threads.

For improving performance with many concurrently opened connections, see "Pooling Connections" on page 3-3.

### Application-Managed Data Buffering

When you provide data for bind parameters by the setxxx methods in parameterized statements, the values are copied into an internal data buffer, and the copied values are then provided to the database server for insertion. To reduce overhead of copying string type data that is available in user buffers, use the setDataBuffer() and next() methods of the ResultSet Class and the execute() method of the Statement Class.
setDataBuffer() Method

For high performance applications, OCCI provides the `setDataBuffer` method whereby the data buffer is managed by the application. The following example shows the `setDataBuffer()` method:

```c
void setDataBuffer(int paramIndex,
    void *buffer,
    Type type,
    sb4 size,
    ub2 *length,
    sb2 *ind = NULL,
    ub2 *rc = NULL);
```

The following parameters are used in the previous method example:

- `paramIndex`: Parameter number
- `buffer`: Data buffer containing data
- `type`: Type of the data in the data buffer
- `size`: Size of the data buffer
- `length`: Current length of data in the data buffer
- `ind`: Indicator information. This indicates whether the data is NULL or not. For parameterized statements, a value of -1 means a NULL value is to be inserted. For data returned from callable statements, a value of -1 means NULL data is retrieved.
- `rc`: Return code. This variable is not applicable to data provided to the `Statement` method. However, for data returned from callable statements, the return code specifies parameter-specific error numbers.

Not all data types can be provided and retrieved by the `setDataBuffer()` method. For instance, C++ Standard Library strings cannot be provided with the `setDataBuffer()` interface.

There is an important difference between the data provided by the `setxxx()` methods and `setDataBuffer()` method. When data is copied in the `setxxx()` methods, the original can change once the data is copied. For example, you can use a `setString(str1)` method, then change the value of `str1` before execute. The value of `str1` that is used is the value at the time `setString(str1)` is called. However, for data provided by the `setDataBuffer()` method, the buffer must remain valid until the execution is completed.

If iterative executes or the `executeArrayUpdate()` method is used, then data for multiple rows and iterations can be provided in a single buffer. In this case, the data for the ith iteration is at `buffer + (i-1) * size` address and the length, indicator, and return codes are at `*(length + i), *(ind + i), and *(rc + i)` respectively.

This interface is also meant for use with array executions and callable statements that have array or `OUT` bind parameters.

The same method is available in the `ResultSet` class to retrieve data without re-allocating the buffer for each fetch.

See Also: Table 5-2, "External Data Types and Corresponding C++ and OCCI Types" in Chapter 5, "Data Types" for specific cases
executeArrayUpdate() Method

If all data is provided with the `setDataBuffer()` methods or output streams (that is, no `setxx()` methods besides `setDataBuffer()` or `getStream()` are called), then there is a simplified way of doing iterative execution.

In this case, you should not call `setMaxIterations()` and `setMaxParamSize()`. Instead, call the `setDataBuffer()` or `getStream()` method for each parameter with the appropriate size arrays to provide data for each iteration, followed by the `executeArrayUpdate(int arrayLength)` method. The `arrayLength` parameter specifies the number of elements provided in each buffer. Essentially, this is same as setting the number of iterations to `arrayLength` and executing the statement.

Since the stream parameters are specified only once, they can be used with array executes as well. However, if any `setxx()` methods are used, then the `addIteration()` method is called to provide data for multiple rows. To compare the two approaches, consider Example 12–2 that inserts two employees in the `employees` table:

**Example 12–2 How to Insert Records Using the addIteration() method**

```
Statement *stmt = conn->createStatement(
    "insert into departments (department_id, department_name) values(:1, :2)\\n"
    char dnames[][100] = {"Community Outreach", "University Recruiting"};
    ub2 dnameLen[2];
    for (int i = 0; i < 2; i++)
        dnameLen[i] = strlen(dnames[i] + 1);
    stmt->setMaxIterations(2); // set maximum number of iterations
    stmt->setInt(1, 7369); // specify data for the first row
    stmt->setDataBuffer(2, dnames, OCI_SQLT_STR, sizeof(dnames[0]), dnameLen);
    stmt->addIteration();
    stmt->setInt(1, 7654); // specify data for the second row
    // a setDataBuffer is unnecessary for the second
    // bind parameter as data provided through
    // setDataBuffer is specified only once.
    stmt->executeUpdate();
```

However, if the first parameter could also be provided through the `setDataBuffer()` interface, then, instead of the `addIteration()` method, you would use the `executeArrayUpdate()` method, as shown in Example 12–3:

**Example 12–3 How to Insert Records Using the executeArrayUpdate() Method**

```
Statement *stmt = conn->createStatement(
    "insert into departments (department_id, department_name) values(:1, :2)\\n"
    char dnames[][100] = {"Community Outreach", "University Recruiting"};
    ub2 dnameLen[2];
    for (int i = 0; i < 2; i++)
        dnameLen[i] = strlen(dnames[i] + 1);
    int ids[2] = {7369, 7654};
    ub2 idLen[2] = {sizeof(ids[0]), sizeof(ids[1])};
    stmt->setDataBuffer(1, ids, OCIINT, sizeof(ids[0]), idLen);
    stmt->setDataBuffer(2, dnames, OCI_SQLT_STR, sizeof(dnames[0]), dnameLen);
```
stmt->executeArrayUpdate(2); // data for two rows is inserted.

Array Fetch Using next() Method

If the application is fetching data with only the setDataBuffer() interface or the stream interface, then an array fetch can be executed. The array fetch is implemented through the next() method of the ResultSet class. You must process the results obtained through next() before calling it again.

Example 12–4  How to use Array Fetch with a ResultSet

```
ResultSet *resultSet = stmt->executeQuery(...);
resultSet->setDataBuffer(...);
while (resultSet->next(numRows) == DATA_AVAILABLE)
    process(resultSet->getNumArrayRows());
```

This causes up to numRows amount of data to be fetched for each column. The buffers specified with the setDataBuffer() interface should large enough to hold at least numRows of data.

Modifying Rows Iteratively

To process batch errors, specify that the Statement object is in a batchMode of execution using the setBatchErrorMode() method. Once the batchMode is set and a batch update runs, any resulting errors are reported through the BatchSQLException Class.

The BatchSQLException class provides methods that handle batch errors. Example 12–5 illustrates how batch handling can be implemented within any OCCI application.

Example 12–5  How to Modify Rows Iteratively and Handle Errors

1. Create the Statement object and set its batch error mode to TRUE.

```
Statement *stmt = conn->createStatement("...");
stmt->setBatchErrorMode(true);
```

2. Perform programmatic changes necessary by the application.

3. Update the statement.

```
try {
    updateCount = stmt->executeUpdate();
}
```

4. Catch and handle any errors generated during the batch insert or update.

```
catch (BatchSQLException &batchEx) {
    cout<"Batch Exception: "<<batchEx.what()<<endl;
    int errCount = batchEx.getFailedRowCount();
    cout << "Number of rows failed " << errCount <<endl;
    for (int i = 0; i < errCount; i++)
    {
        SQLException err = batchEx.getException(i);
        unsigned int rowIndex = batchEx.getRowNum(i);
        cout<"Row " << rowIndex << " failed because of " << err.getErrorCode() <<endl;
    }
}
Run-time Load Balancing of the Stateless Connection Pool

Run-time load balancing in a stateless connection pool dynamically routes connection requests to the least loaded instance of the database. This is achieved by use of service metrics, which are distributed by the Oracle RAC load-balancing advisory.

The feature modifies the stateless connection pool in the following ways:

- The pool receives periodic notifications about the instance load.
- When a request for a connection is received, the pool picks the best possible connection for the type of request, based on the load of the instance.
- The stateless connection pool periodically terminates connections of overloaded instances, maintaining the connection topology that corresponds to the instance load.
- Since the connections to overloaded instances may be terminated, the pool creates new connections to maintain the concurrency requirement. These new connections are created using the connect-time load balancing of the Oracle RAC listener.

Run-time load balancing is turned on by default when the OCCI environment is created in THREADED_MUTEXED and EVENTS modes, and when the server is configured to issue event notifications.

See Also: Oracle Call Interface Programmer’s Guide

API Support

New NO_RLB option for the PoolType attribute of the StatelessConnectionPool Class on page 13-203 disables run-time load balancing.

Fault Diagnosability

Fault diagnosability captures diagnostic data, such as dump files or core dump files, on the OCCI client when a problem incident occurs. For each incident, the fault diagnosability feature creates an Automatic Diagnostic Repository (ADR) subdirectory for storing this diagnostic data. For example, if either a Linux or a UNIX application fails with a null pointer reference, then the core file appears in the ADR home directory (if it exists), not in the operating system directory. This section discusses the ADR subdirectory structure and the utility to manage its output, the ADR Command Interpreter (ADRCI).

An ADR home is the root directory for all diagnostic data for an instance of a product, such as OCCI, and a particular operating system user. All ADR homes appear under the same root directory, the ADR base.

See Also: Oracle Database Administrator’s Guide
ADR Base Location

The location of the ADR base is determined in the following order:

1. In the sqlnet.ora file (on Windows, in the %TNS_ADMIN% directory, or on Linux or UNIX, in the $TNS_ADMIN directory).
   - If there is no TNS_ADMIN directory, then sqlnet.ora is stored in the current directory.
   - If the ADR base is listed in the sqlnet.ora file, it is a statement of the type:
     \[ ADR_BASE=/directory/adr \]
     where:
     - The \textit{adr} argument is a directory that must exist and be writable by all operating system users who execute OCCI applications and want to share the same ADR base.
     - The \textit{directory} argument is the path name.
   - If ADR_BASE is set, and if all users share a single sqlnet.ora file, then OCCI stops searching when directory \textit{adr} does not exist or if it is not writable. If ADR_BASE is not set, then OCCI continues the search, testing for the existence of other specific directories.
   - For example, if sqlnet.ora contains the entry \texttt{ADR_BASE=/home/chuck/test} then:
     - ADR base is:
       \[ /home/chuck/test/oradiag_chuck \]
     - ADR home may be:
       \[ /home/chuck/test/oradiag_chuck/diag/clients/user_chuck/host_4144260688_11 \]

2. If the Oracle base exists (on Windows: %ORACLE_BASE%, or on Linux and UNIX: $ORACLE_BASE), the client subdirectory also exists because it is created by the Oracle Universal Installer when the database is installed.
   - For example, if \texttt{$ORACLE_BASE} is \texttt{/home/chuck/obase}, then:
     - ADR base is:
       \[ /home/chuck/obase \]
     - ADR home may be:
       \[ /home/chuck/obase/diag/clients/user_chuck/host_4144260688_11 \]

3. If the Oracle home exists (on Windows: %ORACLE_HOME%, or on Linux and UNIX: $ORACLE_HOME), the client subdirectory also exists because it is created by the Oracle Universal Installer when the database is installed.
   - For example, if \texttt{$ORACLE_HOME} is \texttt{/ade/chuck_l1/oracle}, then:
     - ADR base is:
       \[ /ade/chuck_l1/oracle/log \]
     - ADR home may be:
       \[ /ade/chuck_l1/oracle/log/diag/clients/user_chuck/host_4144260688_11 \]
4. On the operating system home directory.
   - On Windows, the operating system home directory is %USERPROFILE%.
     The location of folder Oracle is at:
     C:\Documents and Settings\chuck
     If the application runs as a service, the home directory option is skipped.
   - On Linux and UNIX, the operating system home directory is $HOME.
     The location may be:
     /home/chuck/oradiag_chuck

     For example, in an Instant Client, if $HOME is /home/chuck, then:
     - ADR base is:
       /home/chuck/oradiag_chuck
     - ADR home may be:
       /home/chuck/oradiag_chuck/diag/clients/user_chuck/host_4144260688_11

     See Also: "Instant Client" on page 2-2

5. In the temporary directory.
   - On Windows, the temporary directories are searched in the following order:
     - %TMP%
     - %TEMP%
     - %USERPROFILE%
     - Windows system directory
   - On Linux and UNIX, the temporary directory is in /var/tmp.

     For example, in an Instant Client, if $HOME is not writable, then:
     - ADR base is:
       /var/tmp/oradiag_chuck
     - ADR home may be:
       /var/tmp/oradiag_chuck/diag/clients/user_chuck/host_4144260688_11

     If none of these directory choices are available and writable, ADR is not created and diagnostics are not stored.

     See Also: Oracle Database Net Services Reference

Using ADRCI

ADRCI is a command-line tool that enables you to view diagnostic data within the ADR, and to package incident and problem information into a zip file that can be shared with Oracle Support. ADRCI can be used either interactively and through a script.

A problem is a critical error in OCI or the client. Each problem has a problem key. An incident is a single occurrence of a problem, and it is identified by a unique numeric
incident ID. Each incident has a problem key which has a set of attributes: the ORA error number, error parameter values, and similar information. Two incidents have the same root cause if their problem keys match.

The following examples demonstrate how to use ADRCI on a Linux operating system. Note that ARDCI commands are case-insensitive. All user input is in bold typeface.

**Example 12–6  How to Use ADRCI for OCCI Application Incidents**

To launch ADRCI in a Linux system, use the `adrci` command. Once ADRCI starts, find out the particulars of the `show base` command with `help`, and then determine the base of a particular client using the `-product client` option (necessary for OCCI applications). To set the ADRCI base, use the `set base` command. Once ADRCI starts, then the default ADR base is for the rdbms server. The `$ORACLE_HOME` is set to `/ade/chuck_l3/oracle`. To view the incidents, use the `show incidents` command. To exit ADRCI, use the `quit` command.

```%
% adrci
Copyright (c) 1982, 2008, Oracle. All rights reserved.
adrci> help show base
Usage: SHOW BASE [-product <product_name>]
Purpose: Show the current ADR base setting.

Options:
[-product <product_name>]: This option allows users to show the given product's ADR Base location. The current registered products are 'CLIENT' and "ADRCI".

Examples:
show base -product client
show base

adrci> show base -product client
ADR base is '/ade/chuck_l3/oracle/log'

adrci> help set base
Usage: SET BASE <base_str>
Purpose: Set the ADR base to use in the current ADRCI session.
If there are valid ADR homes under the base, all homes are added to the current ADRCI session.

Arguments:
-base_str>: It is the ADR base directory, which is a system-dependent directory path string.

Notes:
On platforms that use "." to signify current working directory, it can be used as base_str.

Example:
set base /net/sttttdl/scratch/someone/view_storage/someone_v1/log
set base .
```
Example 12–7  How to Use ADRCI for Instant Client

Because Instant Client does not use $ORACLE_HOME, the default ADR base is the user’s home directory.

```
adrci> show base -product client
ADR base is "/*/home/chuck/oradiag_chuck"
adrci> set base /home/chuck/oradiag_chuck
```

ADRCI> show incidents

ADR Home = /home/chuck/oradiag_chuck/diag/clients/user_chuck/host_4144260688_11:

<table>
<thead>
<tr>
<th>INCIDENT_ID</th>
<th>PROBLEM_KEY</th>
<th>CREATE_TIME</th>
</tr>
</thead>
</table>

1 rows fetched

```
adrci> quit
```

See Also:
- Oracle Database Utilities for an introduction to the ADRCI
- "Instant Client" on page 2-2

Controlling ADR Creation and Disabling Fault Diagnosability

To disable the fault diagnosability feature, you must turn off the capture of diagnostics. Edit the sqlnet.ora file by changing the values of the DIAG_ADR_ENABLED and DIAG_DDE_ENABLED parameters to either FALSE or OFF; the default values are either TRUE or ON.

To turn off the OCCI signal handler and to re-enable standard operating system failure processing, edit the sqlnet.ora file by adding the corresponding parameter: DIAG_SIGHANDLER_ENABLED=FALSE.

See Also: Oracle Database Net Services Reference

Client Result Cache

The Client Result Cache improves the response times of queries that are executed repeatedly. This feature uses client memory to cache results of SQL queries executed and fetched from the database. Subsequent execution of the same query fetches the results from the client cache, reducing server CPU usage. Because database round-trips are eliminated, applications have improved response times.

OCCI applications may transparently use the Client Result Cache feature by enabling OCCI statement caching. Note that SELECT queries that must be cached are annotated with a /*+ result_cache */ hint. Example 12–8 shows how to create a OCCI Statement object that uses such a SELECT query.

Example 12–8  How to Enable and Use the Client Result Cache

```
Connection *conn;
```
Statement *stmt;
ResultSet *rs;

... //enable OCCI Statement Caching
conn->setStmtCacheSize(20);

//Specify the hint in the SELECT query
stmt = conn->createStatement("select /*+ result_cache */ * from products \
    where product_id = :1");

//the following execute fetches rows from the client cache if
//the query results are cached. If this is the first execute
//of the query, the results fetched from the server are
//cached on the client side.
rs = stmt->executeQuery();

For usage guidelines, cache consistency, and restrictions, see Oracle Call Interface
Programmer’s Guide.
This chapter describes the OCCI classes and methods for C++.

See Also:

- Format Models in Oracle Database SQL Language Reference
- Table A-1 in Oracle Database Globalization Support Guide
Table 13–1 provides a brief description of all the OCCI classes. This section is followed by detailed descriptions of each class and its methods.

Table 13–1 Summary of OCCI Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Class</td>
<td>Represents an agent in the Advanced Queuing context.</td>
</tr>
<tr>
<td>AnyData Class</td>
<td>Provides methods for the Object Type Translator (OTT) utility, read and write SQL methods for linearization of objects, and conversions to and from other data types.</td>
</tr>
<tr>
<td>BatchSQLException Class</td>
<td>Provides methods for handling batch processing errors; extends the SQLException Class.</td>
</tr>
<tr>
<td>Bfile Class</td>
<td>Provides access to a SQL BFILE value.</td>
</tr>
<tr>
<td>Blob Class</td>
<td>Provides access to a SQL BLOB value.</td>
</tr>
<tr>
<td>Bytes Class</td>
<td>Examines individual bytes of a sequence for comparing bytes, searching bytes, and extracting bytes.</td>
</tr>
<tr>
<td>Clob Class</td>
<td>Provides access to a SQL CLOB value.</td>
</tr>
<tr>
<td>Connection Class</td>
<td>Represents a connection with a specific database.</td>
</tr>
<tr>
<td>ConnectionPool Class</td>
<td>Represents a connection pool with a specific database.</td>
</tr>
<tr>
<td>Consumer Class</td>
<td>Supports dequeuing of Messages and controls the dequeuing options.</td>
</tr>
<tr>
<td>Date Class</td>
<td>Specifies abstraction for SQL DATE data items. Also provides formatting and parsing operations to support the OCCI escape syntax for date values.</td>
</tr>
<tr>
<td>Environment Class</td>
<td>Provides an OCCI environment to manager memory and other resources of OCCI objects. An OCCI driver manager maps to an OCCI environment handle.</td>
</tr>
<tr>
<td>IntervalDS Class</td>
<td>Represents a time period in terms of days, hours, minutes, and seconds.</td>
</tr>
<tr>
<td>IntervalYM Class</td>
<td>Represents a time period in terms of year and months.</td>
</tr>
<tr>
<td>Listener Class</td>
<td>Listens on behalf of one or more agents on one or more queues.</td>
</tr>
<tr>
<td>Map Class</td>
<td>Used to store the mapping of the SQL structured type to C++ classes.</td>
</tr>
<tr>
<td>Message Class</td>
<td>A unit that is enqueued or dequeued.</td>
</tr>
<tr>
<td>MetaData Class</td>
<td>Used to determine types and properties of columns in a ResultSet, that of existing schema objects in the database, or the database as a whole.</td>
</tr>
<tr>
<td>NotifyResult Class</td>
<td>Used to hold notification information from the Streams AQ callback function.</td>
</tr>
<tr>
<td>Number Class</td>
<td>Models the numeric data type.</td>
</tr>
</tbody>
</table>
Using OCCI Classes

OCCI classes are defined in the `oracle::oci` namespace. An OCCI class name within the `oracle::oci` namespace can be referred to in one of three ways:

- Use the scope resolution operator (`::`) for each OCCI class name.
- Use the `using` declaration for each OCCI class name.
- Use the `using` directive for all OCCI class name.

### Using Scope Resolution Operator for OCCI

The scope resolution operator (`::`) is used to explicitly specify the `oracle::oci` namespace and the OCCI class name. To declare `myConnection`, a `Connection` object, using the scope resolution operator, you would use the following syntax:

```cpp
oracle::oci::Connection myConnection;
```

### Using Declaration in OCCI

The `using` declaration is used when the OCCI class name can be used in a compilation unit without conflict. To declare the OCCI class name in the `oracle::oci` namespace, you would use the following syntax:

```cpp
using oracle::oci::Connection;
```
Connection now refers to oracle::occi::Connection, and myConnection can be declared as Connection myConnection;

Using Directive in OCCI

The using directive is used when all OCCI class names can be used in a compilation unit without conflict. To declare all OCCI class names in the oracle::occi namespace, you would use the following syntax:

using oracle::occi;

Then, just as with the using declaration, the following declaration would now refer to the OCCI class Connection as Connection myConnection;

Using Advanced Queuing in OCCI

The Advanced Queuing classes Producer, Consumer, Message, Agent, Listener, Subscription and NotifyResult are defined in oracle::occi::aq namespace.

OCCI Support for Windows NT and z/OS

When building OCCI application on Windows, a preprocessor definition for WIN32COMMON has to be added.

The following global methods are designed for accessing collections of Refs in ResultSet Class and Statement Class on Windows NT and z/OS. While method names changed, the number of parameters and their types remain the same.

- Use getVectorOfRefs() instead of getVector() on Windows NT and z/OS.
- Use setVectorOfRefs() instead of setVector() on Windows NT and z/OS.

Applications on Windows NT and z/OS should be calling these new methods only for retrieving and inserting collections of references. Applications not running on Windows NT or z/OS can use either set of accessors. However, Oracle recommends the use of the new methods for any vector operations with Refs.

Working with Collections of Refs

Collections of Refs can be fetched and inserted using methods of the following classes:

ResultSet Class

Fetching Collection of Refs Use the following version of getVectorOfRefs() on page 13-194 to return a column of references:

```cpp
void getVectorOfRefs(
    ResultSet  *rs,
    unsigned int index,
    vector<Ref<T> > &vect);
```

Statement Class

Fetching Collection of Refs Use getVectorOfRefs() on page 13-230 to return a collection of references from a column:

```cpp
void getVectorOfRefs(
    Statement  *stmt,
    unsigned int index,
    vector<Ref<T> > &vect);
```
Inserting a Collection of Refs  Use `setVectorOfRefs()` on page 13-252 to insert a collection of references into a column:

```cpp
template <class T>
void setVectorOfRefs(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<Ref<T> > &vect,
    const string &sqltype);
```

Inserting a Collection of Refs: Multibyte Support  The following method is necessary for multibyte support:

```cpp
void setVectorOfRefs(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<Ref<T> > &vect,
    const string &schemaName,
    const string &typeName);
```

Inserting a Collection of Refs: UString (UTF16) Support  The following method is necessary for UString support:

```cpp
template <class T>
void setVectorOfRefs(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<Ref<T> > &vect,
    const UString &schemaName,
    const UString &typeName);
```
Common OCCI Constants

Table 13–2 defines the common constants used by all OCCI classes. Constants that are defined for use within specific classes are summarized at the beginning of class-specific sections.

Table 13–2  Enumerated Values Used by All OCCI Classes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>LockOptions</td>
<td>- OCCI_LOCK_NONE clears the lock setting on the Ref object.</td>
</tr>
<tr>
<td></td>
<td>- OCCI_LOCK_X indicates that the object should be locked, and to wait for</td>
</tr>
<tr>
<td></td>
<td>the lock to be available if the object is locked by another session.</td>
</tr>
<tr>
<td></td>
<td>- OCCI_LOCK_X_NOWAIT indicates that the object should be locked, and</td>
</tr>
<tr>
<td></td>
<td>returns an error if it is locked by another session.</td>
</tr>
<tr>
<td>CharSetForm</td>
<td>- OCCI_SQLCS_IMPLICIT indicates that the local database character set</td>
</tr>
<tr>
<td></td>
<td>must be used.</td>
</tr>
<tr>
<td></td>
<td>- OCCI_SQLCS_NCHAR indicates that the local database NCHAR set must be</td>
</tr>
<tr>
<td></td>
<td>used.</td>
</tr>
<tr>
<td></td>
<td>- OCCI_SQLCS_EXPLICIT indicates that the character set is specified</td>
</tr>
<tr>
<td></td>
<td>explicitly.</td>
</tr>
<tr>
<td></td>
<td>- OCCI_SQLCS_FLEXIBLE means that the character set is a PL/SQL</td>
</tr>
<tr>
<td></td>
<td>flexible parameter.</td>
</tr>
<tr>
<td>ReturnStatus</td>
<td>- OCCI_SUCCESS indicates that the call has been made successfully</td>
</tr>
<tr>
<td></td>
<td>(transaction failover mode).</td>
</tr>
<tr>
<td></td>
<td>- FO_RETRY indicates that the call should be retried (transaction failover</td>
</tr>
<tr>
<td></td>
<td>mode).</td>
</tr>
</tbody>
</table>
Agent Class

The **Agent** class represents an agent in the Advanced Queuing context.

### Table 13–3  Summary of Agent Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent() on page 13-7</td>
<td>Agent class constructor.</td>
</tr>
<tr>
<td>getAddress() on page 13-7</td>
<td>Returns the address of the <strong>Agent</strong>.</td>
</tr>
<tr>
<td>getName() on page 13-8</td>
<td>Returns the name of the <strong>Agent</strong>.</td>
</tr>
<tr>
<td>getProtocol() on page 13-8</td>
<td>Returns the protocol of the <strong>Agent</strong>.</td>
</tr>
<tr>
<td>isNull() on page 13-8</td>
<td>Tests whether the <strong>Agent</strong> object is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>operator=() on page 13-8</td>
<td>Assignment operator for <strong>Agent</strong>.</td>
</tr>
<tr>
<td>setAddress() on page 13-8</td>
<td>Sets address of the <strong>Agent</strong> object.</td>
</tr>
<tr>
<td>setName() on page 13-9</td>
<td>Sets name of the <strong>Agent</strong> object.</td>
</tr>
<tr>
<td>setNull() on page 13-9</td>
<td>Sets <strong>Agent</strong> object to <strong>NULL</strong>.</td>
</tr>
<tr>
<td>setProtocol() on page 13-9</td>
<td>Sets protocol of the <strong>Agent</strong> object.</td>
</tr>
</tbody>
</table>

### Agent()

Agent class constructor.

#### Syntax Description

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent( const Environment *env);</td>
<td>Creates an <strong>Agent</strong> object initialized to its default values.</td>
</tr>
<tr>
<td>Agent( const Agent&amp; agent);</td>
<td>Copy constructor.</td>
</tr>
<tr>
<td>Agent( const Environment *env, const string&amp; name, const string&amp; address, unsigned int protocol = 0);</td>
<td>Creates an <strong>Agent</strong> object with specified <strong>Agent</strong>'s name, address, and protocol.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>Environment</td>
</tr>
<tr>
<td>name</td>
<td>Name</td>
</tr>
<tr>
<td>agent</td>
<td>Original agent</td>
</tr>
<tr>
<td>address</td>
<td>Address</td>
</tr>
<tr>
<td>protocol</td>
<td>Protocol</td>
</tr>
</tbody>
</table>

### getAddress()

Returns a string containing **Agent**'s address.
getName()

Syntax
string getAddress() const;

getName()

Returns a string containing Agent’s name.

Syntax
string getName() const;

getProtocol()

Syntax
unsigned int getProtocol() const;

getProtocol()

Returns a numeric code representing Agent’s protocol.

isNull()

Tests whether the Agent object is NULL. If the Agent object is NULL, then TRUE is returned; otherwise, FALSE is returned.

Syntax
bool isNull() const;

operator=()

Assignment operator for Agent class.

Syntax
void operator=(
    const Agent& agent);

Parameter Description
agent The original Agent object.

setAddress()

Sets the address of the Agent object.

Syntax
void setAddress(
    const string& addr);

Parameter Description
addr The name of the Agent object.
**setName()**

Sets the name of the Agent object.

**Syntax**

```cpp
void setName(
    const string& name);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the Agent object.</td>
</tr>
</tbody>
</table>

**setNull()**

Sets the Agent object to NULL. Unless operating in an inner scope, this call should be made before terminating the Connection used to create this Agent.

**Syntax**

```cpp
void setNull();
```

**setProtocol()**

Sets the protocol of the Agent object.

**Syntax**

```cpp
void setProtocol(
    unsigned int protocol = 0);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol</td>
<td>The protocol of the Agent object.</td>
</tr>
</tbody>
</table>
The `AnyData` class models self-descriptive data by encapsulating the type information with the actual data. `AnyData` is used primarily with OCCI Advanced Queuing feature, to represent and enqueue data and to receive messages from queues as `AnyData` instances.

Most SQL and user-defined types can be converted into an `AnyData` type using the `setFromxxx()` methods. An `AnyData` object can be converted into most SQL and user-defined types using `getAsxxx()` methods. `SYS.ANYDATA` type models `AnyData` both in SQL and PL/SQL. See Table 13-4, "OCCI Data Types supported by AnyData Class" for supported data types.

The `getType()` call returns the `TypeCode` represented by an `AnyData` object, while the `isNull()` call determines if `AnyData` contains a NULL value. The `setNull()` method sets the value of `AnyData` to NULL.

To use the OCCI `AnyData` type, the environment has to be initiated in `OBJECT` mode.

**Example 13–1 Converting From an SQL Pre-Defined Type To AnyData Type**

This example demonstrates how to convert types from `string` to `AnyData`.

```cpp
Connection *conn;
...
AnyData any(conn);
string str("Hello World");
any.setFromString(str);
...
```

**Example 13–2 Creating an SQL Pre-Defined Type From AnyData Type**

This example demonstrates how to convert an `AnyData` object back to a `string` object. Note the use of `getType()` and `isNull()` methods to validate `AnyData` before conversion.

```cpp
Connection *conn;
string str;
...
if (!any.isNull())
{  if (any.getType()==OCCI_TYPECODE_VARCHAR2)
   { str = any.getAsString();
     cout<<str;
   }
}
...
```

**Example 13–3 Converting From a User-Defined Type To AnyData Type**

This example demonstrates how to convert from a user-defined type to `AnyData` type.

```cpp
Connection *conn;
...
// Assume an OBJECT of type Person with the following defined fields
// CREATE TYPE person as OBJECT {
//     FRIST_NAME VARCHAR2(20),
//     LAST_NAME VARCHAR2(25),
//     EMAIL VARCHAR2(25),
//     SALARY NUMBER(8,2)
```
AnyData Class

// Assure relevant classes have been generated by OTT.
...
Person *pers = new Person( "Steve", "Addams",
  "steve.addams@anycompany.com", 50000.00);
AnyData anyObj(conn);
anyObj.setFromObject(pers);
...

Example 13–4 Converting From a User-Defined Type To AnyData Type

This example demonstrates how to convert an AnyData object back to a user-defined type. Note the use of `getType()` and `isNull()` methods to validate AnyData before conversion.

Connection *conn;
// Assume an OBJECT of type Person with the following defined fields
// CREATE TYPE person as OBJECT {
  //  FRIST_NAME VARCHAR2(20),
  //  LAST_NAME VARCHAR2(25),
  //  EMAIL VARCHAR2(25),
  //  SALARY NUMBER(8,2)
  // };
// Assume relevant classes have been generated by OTT.
Person *pers = new Person();
...
If(!anyObj.isNull())
{  if(anyObj.getType()==OCCI_TYPECODE_OBJECT)
      pers = anyObj.getAsObject();
}
...

<table>
<thead>
<tr>
<th>Data Type</th>
<th>TypeCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDouble</td>
<td>OCCI_TYPECODE_BDOUBLE</td>
</tr>
<tr>
<td>BFile</td>
<td>OCCI_TYPECODE_BFILE</td>
</tr>
<tr>
<td>BFloat</td>
<td>OCCI_TYPECODE_BFLOAT</td>
</tr>
<tr>
<td>bytes</td>
<td>OCCI_TYPECODE_RAW</td>
</tr>
<tr>
<td>Date</td>
<td>OCCI_TYPECODE_DATE</td>
</tr>
<tr>
<td>IntervalDS</td>
<td>OCCI_TYPECODE_INTERVAL_DS</td>
</tr>
<tr>
<td>IntervalYM</td>
<td>OCCI_TYPECODE_INTERVAL_YM</td>
</tr>
<tr>
<td>Number</td>
<td>OCCI_TYPECODE_NUMBERB</td>
</tr>
<tr>
<td>PObject</td>
<td>OCCI_TYPECODE_OBJECT</td>
</tr>
<tr>
<td>Ref</td>
<td>OCCI_TYPECODE_REF</td>
</tr>
<tr>
<td>string</td>
<td>OCCI_TYPECODE_VARCHAR2</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>OCCI_TYPECODE_TIMESTAMP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnyData() on page 13-12</td>
<td>AnyData class constructor.</td>
</tr>
</tbody>
</table>
AnyData()

AnyData constructor.

**Syntax**

```cpp
AnyData(
    const Connection *conn);
```
getAsBDouble()

Converts an AnyData object into BDouble.

Syntax
BDouble getAsBDouble() const;

getAsBfile()

Converts an AnyData object into Bfile.

Syntax
Bfile getAsBfile() const;

getAsBFloat()

Converts an AnyData object into BFloat.

Syntax
BFloat getAsBFloat() const;

getAsBytes()

Converts an AnyData object into Bytes.

Syntax
Bytes getAsBytes() const;

getAsDate()

Converts an AnyData object into Date.

Syntax
Date getAsDate() const;

getAsIntervalDS()

Converts an AnyData object into IntervalDS.

Syntax
IntervalDS getAsIntervalDS() const;
getAsIntervalYM()

Converts an AnyData object into IntervalYM.

Syntax
IntervalYS getAsIntervalYM() const;

getAsNumber()

Converts an AnyData object into Number.

Syntax
Number getAsNumber() const;

getAsObject()

Converts an AnyData object into PObject.

Syntax
PObject* getAsObject() const;

getAsRef()

Converts an AnyData object into RefAny.

Syntax
RefAny getAsRef() const;

getAsString()

Converts an AnyData object into a namespace string.

Syntax
string getAsString() const;

getAsTimestamp()

Converts an AnyData object into Timestamp.

Syntax
Timestamp getAsTimestamp() const;

getType()

Retrieves the data type held by the AnyData object. Refer to Table 13-4 on page 13-11 for valid values for TypeCode.
Syntax

TypeCode getType();

isNull()

Tests whether the AnyData object is NULL. If the AnyData object is NULL, then TRUE is returned; otherwise, FALSE is returned.

Syntax

bool isNull() const;

setFromBDouble()

Converts a BDouble into AnyData.

Syntax

void setFromBDouble(
    const BDouble& bdouble);

Parameter Description
bdouble The BDouble that is converted into AnyData.

setFromBFile()

Converts a Bfile into AnyData.

Syntax

void setFromBfile(
    const Bfile& bfile);

Parameter Description
bfile The Bfile that is converted into AnyData.

setFromBFloat()

Converts a BFloat into AnyData.

Syntax

void setFromBFloat(
    const BFloat& bfloat);

Parameter Description
bfloat The BFloat that is converted into AnyData.
setFromBytes()

Converts a Bytes into AnyData.

**Syntax**

```cpp
void setFromBytes(
    const Bytes& bytes);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>The Bytes that is converted into AnyData.</td>
</tr>
</tbody>
</table>

setFromDate()

Converts a Date into AnyData.

**Syntax**

```cpp
void setFromDate(
    const Date& date);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>The Date that is converted into AnyData.</td>
</tr>
</tbody>
</table>

setFromIntervalDS()

Converts an IntervalDS into AnyData.

**Syntax**

```cpp
void setFromIntervalDS(
    const IntervalDS& intervalds);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intervalds</td>
<td>The IntervalDS that is converted into AnyData.</td>
</tr>
</tbody>
</table>

setFromIntervalYM()

Converts an IntervalYM into AnyData.

**Syntax**

```cpp
void setFromIntervalYM(
    const IntervalYM& intervalym);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intervalym</td>
<td>The IntervalYM that is converted into AnyData.</td>
</tr>
</tbody>
</table>
setFromNumber()

Converts a Number into AnyData.

**Syntax**

```cpp
void setFromNumber(
    const Number& num);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>The Number that is converted into AnyData.</td>
</tr>
</tbody>
</table>

setFromObject()

Converts a PObject into AnyData.

**Syntax**

```cpp
void setFromObject(
    const PObject* objptr);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objptr</td>
<td>The PObject that is converted into AnyData.</td>
</tr>
</tbody>
</table>

setFromRef()

Converts a PObject into AnyData.

**Syntax**

```cpp
void setFromRef(
    const RefAny& ref
    const string &typeName,
    const string &schema);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref</td>
<td>The RefAny that is converted into AnyData.</td>
</tr>
<tr>
<td>typeName</td>
<td>The name of the type.</td>
</tr>
<tr>
<td>schema</td>
<td>The name of the schema where the type is defined.</td>
</tr>
</tbody>
</table>

setFromString()

Converts a namespace string into AnyData.

**Syntax**

```cpp
void setFromString(
    string& str);
```
### setFromTimestamp()

Converts a `Timestamp` into `AnyData`.

**Syntax**

```c++
void setFromTimestamp(
    const Timestamp& timestamp);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str</code></td>
<td>The namespace string that is converted into <code>AnyData</code>.</td>
</tr>
</tbody>
</table>

### setNull()

Sets `AnyData` object to NULL.

**Syntax**

```c++
void setNull();
```
BatchSQLException Class

The BatchSQLException class provides methods for handling batch processing errors. Because BatchSQLException class is derived from the SQLException Class, all BatchSQLException instances support all methods of SQLException, in addition to the methods summarized in Table 13–6.

See Also: "Modifying Rows Iteratively" section in Example 12, "Optimizing Performance of OCCI Applications".

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>getException() on page 13-19</td>
<td>Returns the exception.</td>
</tr>
<tr>
<td>getFailedRowCount() on page 13-19</td>
<td>Returns the number of rows with failed inserts or updates.</td>
</tr>
<tr>
<td>getRowNum() on page 13-19</td>
<td>Returns the number of the row that has an insert or updated error</td>
</tr>
</tbody>
</table>

**getException()**

Returns the exception that matches the specified index.

**Syntax**

```cpp
SQLException getSQLException ( 
    unsigned int index) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index into the list of errors returned by the batch process.</td>
</tr>
</tbody>
</table>

**getFailedRowCount()**

Returns the number of rows for which the statement insert or update failed.

**Syntax**

```cpp
unsigned int getFailedRowCount( ) const;
```

**getRowNum()**

Returns the number of the row with an error, matching the specified index.

**Syntax**

```cpp
unsigned int getRowNum( 
    unsigned int index) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index into the list of errors returned by the batch process.</td>
</tr>
</tbody>
</table>
The `BFILE` class defines the common properties of objects of type `BFILE`. A `BFILE` is a large binary file stored in an operating system file outside of the Oracle database. A `BFILE` object contains a logical pointer to a `BFILE`, not the `BFILE` itself.

Methods of the `Bfile` class enable you to perform specific tasks related to `Bfile` objects.

Methods of the `ResultSet` and `Statement` classes, such as `getBfile()` and `setBfile()`, enable you to access an SQL `BFILE` value.

The only methods valid on a NULL `Bfile` object are `setName()`, `isNull()`, and `operator()`. A `Bfile` object can be initialized by:

- The `setName()` method. The `BFILE` can then be modified by inserting this `BFILE` into the table and then retrieving it using `SELECT ... FOR UPDATE`. The `write()` method modifies the `BFILE`; however, the modified data is flushed to the table only when the transaction is committed. Note that an `INSERT` operation is not required.

- Assigning an initialized `Bfile` object to it.

**See Also:** In-depth discussion of LOBs in the introductory chapter of *Oracle Database SecureFiles and Large Objects Developer’s Guide*.

### Table 13–7 Summary of Bfile Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Bfile()</code> on page 13-21</td>
<td><code>Bfile</code> class constructor.</td>
</tr>
<tr>
<td><code>close()</code> on page 13-21</td>
<td>Closes a previously opened <code>BFILE</code>.</td>
</tr>
<tr>
<td><code>closeStream()</code> on page 13-21</td>
<td>Closes the stream obtained from the <code>BFILE</code>.</td>
</tr>
<tr>
<td><code>fileExists()</code> on page 13-22</td>
<td>Tests whether the <code>BFILE</code> exists.</td>
</tr>
<tr>
<td><code>getDirAlias()</code> on page 13-22</td>
<td>Returns the directory object of the <code>BFILE</code>.</td>
</tr>
<tr>
<td><code>getFileName()</code> on page 13-22</td>
<td>Returns the name of the <code>BFILE</code>.</td>
</tr>
<tr>
<td><code>getStream()</code> on page 13-22</td>
<td>Returns data from the <code>BFILE</code> as a <code>Stream</code> object.</td>
</tr>
<tr>
<td><code>getUStringDirAlias()</code> on page 13-22</td>
<td>Returns a <code>UString</code> containing the directory object associated with the <code>BFILE</code>.</td>
</tr>
<tr>
<td><code>getUStringFileName()</code> on page 13-23</td>
<td>Returns a <code>UString</code> containing the file name associated with the <code>BFILE</code>.</td>
</tr>
<tr>
<td><code>isInitialized()</code> on page 13-23</td>
<td>Tests whether the <code>Bfile</code> object is initialized.</td>
</tr>
<tr>
<td><code>isNull()</code> on page 13-23</td>
<td>Tests whether the <code>Bfile</code> object is atomically NULL.</td>
</tr>
<tr>
<td><code>isOpen()</code> on page 13-23</td>
<td>Tests whether the <code>BFILE</code> is open.</td>
</tr>
<tr>
<td><code>length()</code> on page 13-23</td>
<td>Returns the number of bytes in the <code>BFILE</code>.</td>
</tr>
<tr>
<td><code>open()</code> on page 13-24</td>
<td>Opens the <code>BFILE</code> with read-only access.</td>
</tr>
<tr>
<td><code>operator=</code> on page 13-24</td>
<td>Assigns a <code>BFILE</code> locator to the <code>Bfile</code> object.</td>
</tr>
<tr>
<td><code>operator==()</code> on page 13-24</td>
<td>Tests whether two <code>Bfile</code> objects are equal.</td>
</tr>
</tbody>
</table>
**Bfile Class**

**Bfile()**

Bfile class constructor.

**close()**

Closes a previously opened Bfile.

**Syntax**

```c
void close();
```

**closeStream()**

Closes the stream obtained from the Bfile.

**Syntax**

```c
void closeStream(
    Stream *stream);
```

**Table 13–7 (Cont.) Summary of Bfile Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator!=()</td>
<td>Tests whether two Bfile objects are not equal.</td>
</tr>
<tr>
<td>operator==()</td>
<td>Reads a specified portion of the BFILE into a buffer.</td>
</tr>
<tr>
<td>setName()</td>
<td>Sets the directory object and file name of the BFILE.</td>
</tr>
<tr>
<td>setNull()</td>
<td>Sets the Bfile object to atomically NULL.</td>
</tr>
</tbody>
</table>

**Syntax Description**

- **Bfile();**
  - Creates a NULL Bfile object.
- **Bfile(const Connection *connectionp);**
  - Creates an uninitialized Bfile object.
- **Bfile(const Bfile &srcBfile);**
  - Creates a copy of a Bfile object.

**Parameter Description**

- **connectionp**
  - The connection pointer
- **srcBfile**
  - The source Bfile object

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>The stream to be closed.</td>
</tr>
</tbody>
</table>
fileExists()

Tests whether the BFILE exists. If the BFILE exists, then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
tbool fileExists() const;
```

getDirAlias()

Returns a string containing the directory object associated with the BFILE.

**Syntax**

```cpp
string getDirAlias() const;
```

getFileName()

Returns a string containing the file name associated with the BFILE.

**Syntax**

```cpp
string getFileName() const;
```

getStream()

Returns a Stream object read from the BFILE. If a stream is open, it is disallowed to open another stream on the BFILE object. The stream must be closed before performing any BFILE object operations.

**Syntax**

```cpp
Stream* getStream(
    unsigned int offset = 1,
    unsigned int amount = 0);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>The starting position at which to begin reading data from the BFILE. If offset is not specified, the data is written from the beginning of the BLOB. Valid values are numbers greater than or equal to 1.</td>
</tr>
<tr>
<td>amount</td>
<td>The total number of bytes to be read from the BFILE; if amount is 0, the data is read in a streamed mode from input offset until the end of the BFILE.</td>
</tr>
</tbody>
</table>

getUStringDirAlias()

Returns a UString containing the directory object associated with the BFILE. Note the UString object is in UTF16 character set. The environment associated with BFILE should be associated with UTF16 characterset.

**Syntax**

```cpp
UString getUStringDirAlias() const;
```
getUStringFileName()

Returns a UString containing the file name associated with the BFILE. Note the UString object is in UTF16 characterset. The environment associated with BFILE should be associated with UTF16 characterset.

Syntax
UString getUStringFileName() const;

isInitialized()

Tests whether the Bfile object has been initialized. If the Bfile object has been initialized, then TRUE is returned; otherwise, FALSE is returned.

Syntax
bool isInitialized() const;

isNull()

Tests whether the Bfile object is atomically NULL. If the Bfile object is atomically NULL, then TRUE is returned; otherwise, FALSE is returned.

Syntax
bool isNull() const;

isOpen()

Tests whether the BFILE is open. The BFILE is considered to be open only if it was opened by a call on this Bfile object. (A different Bfile object could have opened this file as multiple open() calls can be performed on the same file by associating the file with different Bfile objects). If the BFILE is open, then TRUE is returned; otherwise, FALSE is returned.

Syntax
bool isOpen() const;

length()

Returns the number of bytes (inclusive of the end of file marker) in the BFILE.

Syntax
unsigned int length() const;
open()

Opens an existing BFILE for read-only access. This function is meaningful the first time it is called for a BFILE object.

**Syntax**

```cpp
void open();
```

operator=()

Assigns a BFILE object to the current BFILE object. The source BFILE object is assigned to this BFILE object only when this BFILE object gets stored in the database.

**Syntax**

```cpp
BFILE& operator=(
    const BFILE &srcBfile);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcBfile</td>
<td>The BFILE object to be assigned to the current BFILE object.</td>
</tr>
</tbody>
</table>

operator==()

Compares two BFILE objects for equality. The BFILE objects are equal if they both refer to the same BFILE. If the BFILE objects are NULL, then FALSE is returned. If the BFILE objects are equal, then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool operator==(const BFILE &srcBfile) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcBfile</td>
<td>The BFILE object to be compared with the current BFILE object.</td>
</tr>
</tbody>
</table>

operator!=(

Compares two BFILE objects for inequality. The BFILE objects are equal if they both refer to the same BFILE. If the BFILE objects are not equal, then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool operator!=(const BFILE &srcBfile) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcBfile</td>
<td>The BFILE object to be compared with the current BFILE object.</td>
</tr>
</tbody>
</table>
**read()**

Reads a part or all of the `BFILE` into the buffer specified, and returns the number of bytes read.

**Syntax**

```c
unsigned int read(
    unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1) const;
```

**Parameter Description**

- **amt**: The number of bytes to be read. Valid values are numbers greater than or equal to 1.
- **buffer**: The buffer that the `BFILE` data is to be read into. Valid values are numbers greater than or equal to `amt`.
- **bufsize**: The size of the buffer that the `BFILE` data is to be read into. Valid values are numbers greater than or equal to `amt`.
- **offset**: The starting position at which to begin reading data from the `BFILE`. If `offset` is not specified, the data is written from the beginning of the `BFILE`.

**setName()**

Sets the directory object and file name of the `BFILE`.

**Syntax**

```c
void setName(
    const string &dirAlias,
    const string &fileName);
```

**Parameter Description**

- **dirAlias**: The directory object to be associated with the `BFILE`.
- **fileName**: The file name to be associated with the `BFILE`.

**setNull()**

Sets the `Bfile` object to atomically NULL.

**Syntax**

```c
void setNull();
```
Blob Class

The Blob class defines the common properties of objects of type BLOB. A BLOB is a large binary object stored as a column value in a row of a database table. A Blob object contains a logical pointer to a BLOB, not the BLOB itself.

Methods of the Blob class enable you to perform specific tasks related to Blob objects. Methods of the ResultSet and Statement classes, such as getBlob() and setBlob(), enable you to access an SQL BLOB value.

The only methods valid on a NULL Blob object are setName(), isNull(), and operator=().

An uninitialized Blob object can be initialized by:

- The setEmpty() method. The BLOB can then be modified by inserting this BLOB into the table and then retrieving it using SELECT...FOR UPDATE. The write() method modifies the BLOB; however, the modified data is flushed to the table only when the transaction is committed. Note that an update is not required.
- Assigning an initialized Blob object to it.

See Also:
- In-depth discussion of LOBs in Oracle Database SecureFiles and Large Objects Developer’s Guide

Table 13–8 Summary of Blob Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob() on page 13-27</td>
<td>Blob class constructor.</td>
</tr>
<tr>
<td>append() on page 13-27</td>
<td>Appends a specified BLOB to the end of the current BLOB.</td>
</tr>
<tr>
<td>close() on page 13-27</td>
<td>Closes a previously opened BLOB.</td>
</tr>
<tr>
<td>closeStream() on page 13-28</td>
<td>Closes the Stream object obtained from the BLOB.</td>
</tr>
<tr>
<td>copy() on page 13-28</td>
<td>Copies a specified portion of a BFILE or BLOB into the current BLOB.</td>
</tr>
<tr>
<td>getChunkSize() on page 13-28</td>
<td>Returns the smallest data size to perform efficient writes to the BLOB.</td>
</tr>
<tr>
<td>getContentType() on page 13-29</td>
<td>Returns the content type of the Blob.</td>
</tr>
<tr>
<td>getOptions() on page 13-29</td>
<td>Returns the BLOB's LobOptionValue for a specified LobOptionType.</td>
</tr>
<tr>
<td>getStream() on page 13-29</td>
<td>Returns data from the BLOB as a Stream object.</td>
</tr>
<tr>
<td>isOpen() on page 13-30</td>
<td>Tests whether the Blob object is open.</td>
</tr>
<tr>
<td>isNull() on page 13-30</td>
<td>Tests whether the Blob object is atomically NULL.</td>
</tr>
<tr>
<td>length() on page 13-30</td>
<td>Returns the number of bytes in the BLOB.</td>
</tr>
<tr>
<td>open() on page 13-30</td>
<td>Opens the BLOB with read or read/write access.</td>
</tr>
<tr>
<td>operator=() on page 13-30</td>
<td>Assigns a BLOB locator to the Blob object.</td>
</tr>
<tr>
<td>operator==() on page 13-31</td>
<td>Tests whether two Blob objects are equal.</td>
</tr>
</tbody>
</table>
Blob()  

Blob class constructor.

Syntax Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectionp</td>
<td>The connection pointer</td>
</tr>
<tr>
<td>srcBlob</td>
<td>The source Blob object.</td>
</tr>
</tbody>
</table>

append()  

Appends a BLOB to the end of the current BLOB.

Syntax

```c++
void append(
    const Blob &srcBlob);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcBlob</td>
<td>The Blob object to be appended to the current BLOB object.</td>
</tr>
</tbody>
</table>

close()  

Closes a BLOB.
Syntax
void close();

closeStream()

Closes the Stream object obtained from the BLOB.

Syntax
void closeStream(
    Stream *stream);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>The Stream to be closed.</td>
</tr>
</tbody>
</table>

copy()

Copies a part or all of a BFILE or BLOB into the current BLOB.

Syntax
void copy(
    const Bfile &srcBfile,
    unsigned int numBytes,
    unsigned int dstOffset = 1,
    unsigned int srcOffset = 1);

void copy(
    const Blob &srcBlob,
    unsigned int numBytes,
    unsigned int dstOffset = 1,
    unsigned int srcOffset = 1);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcBfile</td>
<td>The BFILE from which the data is to be copied.</td>
</tr>
<tr>
<td>srcBlob</td>
<td>The BLOB from which the data is to be copied.</td>
</tr>
<tr>
<td>numBytes</td>
<td>The number of bytes to be copied from the source BFILE or BLOB. Valid values are numbers greater than 0.</td>
</tr>
<tr>
<td>dstOffset</td>
<td>The starting position at which to begin writing data into the current BLOB. Valid values are numbers greater than or equal to 1.</td>
</tr>
<tr>
<td>srcOffset</td>
<td>The starting position at which to begin reading data from the source BFILE or BLOB. Valid values are numbers greater than or equal to 1.</td>
</tr>
</tbody>
</table>

getChunkSize()

Returns the smallest data size to perform efficient writes to the BLOB.
getChunkSize()

Syntax
unsigned int getChunkSize() const;

getContentType()

Returns the content type of the Blob. If a content type has not been assigned, returns a NULL string.

Syntax
string getContentType();

getOptions()

Returns the BLOB's LobOptionValue for a specified LobOptionType.

Throws an exception if attempting to retrieve a value for an option that is not configured on the database column or partition that stores the BLOB.

Syntax
LobOptionValue getOptions(
    LobOptionType optType);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optType</td>
<td>The LobOptionType setting requested. These may be combined using bitwise or (</td>
</tr>
</tbody>
</table>

getStream()

Returns a Stream object from the BLOB. If a stream is open, it is disallowed to open another stream on Blob object, so the user must always close the stream before performing any Blob object operations.

Syntax
Stream* getStream(
    unsigned int offset = 1,
    unsigned int amount = 0);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>The starting position at which to begin reading data from the BLOB. If offset is not specified, the data is written from the beginning of the BLOB. Valid values are numbers greater than or equal to 1.</td>
</tr>
<tr>
<td>amount</td>
<td>The total number of bytes to be read from the BLOB; if amount is 0, the data is read in a streamed mode from input offset until the end of the BLOB.</td>
</tr>
</tbody>
</table>

isInitialized()

Tests whether the Blob object is initialized. If the Blob object is initialized, then TRUE is returned; otherwise, FALSE is returned.
### isNull()

Tests whether the Blob object is atomically NULL. If the Blob object is atomically NULL, then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool isInitialized() const;
```

### isOpen()

Tests whether the BLOB is open. If the BLOB is open, then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool isOpen() const;
```

### length()

Returns the number of bytes in the BLOB.

**Syntax**

```cpp
unsigned int length() const;
```

### open()

Opens the BLOB in read/write or read-only mode.

**Syntax**

```cpp
void open(
    LobOpenMode mode = OCCI_LOB_READWRITE);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mode</code></td>
<td>The mode the BLOB is to be opened in. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>- OCCI_LOB_READWRITE</td>
</tr>
<tr>
<td></td>
<td>- OCCI_LOB_READONLY</td>
</tr>
</tbody>
</table>

### operator=()

Assigns a BLOB to the current BLOB. The source BLOB gets copied to the destination BLOB only when the destination BLOB gets stored in the table.

**Syntax**

```cpp
Blob& operator=(
    const Blob &srcBlob);
```
Blob Class

operator==()

Compares two Blob objects for equality. Two Blob objects are equal if they both refer to the same BLOB. Two NULL Blob objects are not considered equal. If the Blob objects are equal, then TRUE is returned; otherwise, FALSE is returned.

Syntax

```cpp
bool operator==(
    const Blob &srcBlob) const;
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcBlob</td>
<td>The source BLOB from which to copy data.</td>
</tr>
</tbody>
</table>

operator!=()

Compares two Blob objects for inequality. Two Blob objects are equal if they both refer to the same BLOB. Two NULL Blob objects are not considered equal. If the Blob objects are not equal, then TRUE is returned; otherwise, FALSE is returned.

Syntax

```cpp
bool operator!=(
    const Blob &srcBlob) const;
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcBlob</td>
<td>The source BLOB to be compared with the current BLOB.</td>
</tr>
</tbody>
</table>

read()

Reads a part or all of the BLOB into a buffer. The actual number of bytes read is returned.

Syntax

```cpp
unsigned int read(
    unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1) const;
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amt</td>
<td>The number of bytes to be read. Valid values are numbers greater than or equal to 1.</td>
</tr>
<tr>
<td>buffer</td>
<td>The buffer that the BLOB data is to be read into. Valid values are numbers greater than or equal to amt.</td>
</tr>
</tbody>
</table>
**setContentType()**

Sets the content type of the Blob. If the Blob is not a SecureFile, throws an exception.

**Syntax**

```cpp
void setContentType(
    const string contenttype);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contenttype</td>
<td>The content type of the Blob; an ASCII Mime compliant string.</td>
</tr>
</tbody>
</table>

**setEmpty()**

Sets the Blob object to empty.

**Syntax**

```cpp
void setEmpty();
void setEmpty(
    const Connection* connectionp);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectionp</td>
<td>The new connection pointer for the BLOB object.</td>
</tr>
</tbody>
</table>

**setNull()**

Sets the Blob object to atomically NULL.

**Syntax**

```cpp
void setNull();
```

**setOptions()**

Specifies a LobOptionValue for a particular LobOptionType. Enables advanced compression, encryption and deduplication of BLOBS. See Table 7–1, “Values of Type LobOptionType” and Table 7–2, “Values of Type LobOptionValue”.

Throws an exception if attempting to set or un-set an option that is not configured on the database column or partition that stores the BLOB.
Throws an exception if attempting to turn off encryption in an encrypted BLOB column.

**Syntax**

```java
void setOptions(
    LobOptionType optType,
    LobOptionValue optValue);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optType</td>
<td>The LobOptionType setting being specified. These may be combined using bitwise OR (</td>
</tr>
<tr>
<td>optValue</td>
<td>The LobOptionValue setting for the LobOptionType specified by the optType parameter</td>
</tr>
</tbody>
</table>

**trim()**

Truncates the BLOB to the new length specified.

**Syntax**

```java
void trim(
    unsigned int newlen);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newlen</td>
<td>The new length of the BLOB. Valid values are numbers less than or equal to the current length of the BLOB.</td>
</tr>
</tbody>
</table>

**write()**

Writes data from a buffer into a BLOB. This method implicitly opens the BLOB, copies the buffer into the BLOB, and implicitly closes the BLOB. If the BLOB is open, use writeChunk() instead. The actual number of bytes written is returned.

**Syntax**

```java
unsigned int write(
    unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amt</td>
<td>The number of bytes to be written to the BLOB.</td>
</tr>
<tr>
<td>buffer</td>
<td>The buffer containing the data to be written to the BLOB.</td>
</tr>
<tr>
<td>bufsize</td>
<td>The size of the buffer containing the data to be written to the BLOB. Valid values are numbers greater than or equal to amt.</td>
</tr>
<tr>
<td>offset</td>
<td>The starting position at which to begin writing data into the BLOB. If offset is not specified, the data is written from the beginning of the BLOB. Valid values are numbers greater than or equal to 1.</td>
</tr>
</tbody>
</table>
writeChunk()

Writes data from a buffer into a previously opened BLOB. The actual number of bytes written is returned.

**Syntax**

```c
unsigned int writeChunk(
    unsigned int amount,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amt</td>
<td>The number of bytes to be written to the BLOB.</td>
</tr>
<tr>
<td>buffer</td>
<td>The buffer containing the data to be written to the BLOB.</td>
</tr>
<tr>
<td>bufsize</td>
<td>The size of the buffer containing the data to be written to the BLOB. Valid values are numbers greater than or equal to <code>amt</code>.</td>
</tr>
<tr>
<td>offset</td>
<td>The starting position at which to begin writing data into the BLOB. If <code>offset</code> is not specified, the data is written from the beginning of the BLOB. Valid values are numbers greater than or equal to 1.</td>
</tr>
</tbody>
</table>
Bytes Class

Methods of the Bytes class enable you to perform specific tasks related to Bytes objects.

Table 13–9  Summary of Bytes Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes() on page 13-35</td>
<td>Bytes class constructor.</td>
</tr>
<tr>
<td>byteAt() on page 13-35</td>
<td>Returns the byte at the specified position of the Bytes object.</td>
</tr>
<tr>
<td>getBytes() on page 13-36</td>
<td>Returns a byte array from the Bytes object.</td>
</tr>
<tr>
<td>isNull() on page 13-36</td>
<td>Tests whether the Bytes object is NULL.</td>
</tr>
<tr>
<td>length() on page 13-36</td>
<td>Returns the number of bytes in the Bytes object.</td>
</tr>
<tr>
<td>operator=() on page 13-36</td>
<td>Assignment operator for Bytes class.</td>
</tr>
<tr>
<td>setNull() on page 13-37</td>
<td>Sets the Bytes object to NULL.</td>
</tr>
</tbody>
</table>

Bytes()

Bytes class constructor.

Syntax class constructor.

```
Bytes()
Environment *env = NULL);
```

```
Bytes(
    unsigned char *value,
    unsigned int count
    unsigned int offset = 0,
    const Environment *env = NULL);
```

```
Bytes(
    const Bytes &e);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>Environment</td>
</tr>
<tr>
<td>value</td>
<td>Initial value of the new object</td>
</tr>
<tr>
<td>count</td>
<td>The size of the subset of the character array that is copied into the new bytes object</td>
</tr>
<tr>
<td>offset</td>
<td>The first position from which to begin copying the character array</td>
</tr>
<tr>
<td>e</td>
<td>The source Bytes object</td>
</tr>
</tbody>
</table>

byteAt()

Returns the byte at the specified position in the Bytes object.
**getBytes()**

Covers bytes from a `Bytes` object into the specified byte array.

**Syntax**
```
void getBytes(
    unsigned char *dst,
    unsigned int count,
    unsigned int srcBegin = 0,
    unsigned int dstBegin = 0) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dst</td>
<td>The destination buffer into which data from the <code>Bytes</code> object is to be written.</td>
</tr>
<tr>
<td>count</td>
<td>The number of bytes to copy.</td>
</tr>
<tr>
<td>srcBegin</td>
<td>The starting position at which data is to be read from the <code>Bytes</code> object; the position of the first byte in the <code>Bytes</code> object is at 0.</td>
</tr>
<tr>
<td>dstBegin</td>
<td>The starting position at which data is to be written in the destination buffer; the position of the first byte in <code>dst</code> is at 0.</td>
</tr>
</tbody>
</table>

**isNull()**

Tests whether the `Bytes` object is atomically NULL. If the `Bytes` object is atomically NULL, then `TRUE` is returned; otherwise `FALSE` is returned.

**Syntax**
```
bool isNull() const;
```

**length()**

This method returns the length of the `Bytes` object.

**Syntax**
```
unsigned int length() const;
```

**operator=( )**

Assignment operator for `Bytes` class.

**Syntax**
```
void operator=(
```
const Bytes& bytes);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>The original Bytes.</td>
</tr>
</tbody>
</table>

**setNull()**

This method sets the Bytes object to atomically NULL.

**Syntax**

```cpp
void setNull();
```
The `Clob` class defines the common properties of objects of type `CLOB`. A `Clob` is a large character object stored as a column value in a row of a database table. A `Clob` object contains a logical pointer to a `CLOB`, not the `CLOB` itself.

Methods of the `Clob` class enable you to perform specific tasks related to `Clob` objects, including methods for getting the length of a SQL `CLOB`, for materializing a `Clob` on the client, and for extracting a part of the `CLOB`.

The only methods valid on a NULL `CLOB` object are `setName()`, `isNull()`, and `operator=()`.

Methods in the `ResultSet` and `Statement` classes, such as `getClob()` and `setClob()`, enable you to access an SQL `CLOB` value.

An uninitialized `CLOB` object can be initialized by:

- The `setEmpty()` method. The `CLOB` can then be modified by inserting this `CLOB` into the table and retrieving it using `SELECT...FOR UPDATE`. The `write()` method modifies the `CLOB`; however, the modified data is flushed to the table only when the transaction is committed. Note that an `insert` is not required.

- Assigning an initialized `Clob` object to it.

**See Also:**

- In-depth discussion of LOBs in the introductory chapter of *Oracle Database SecureFiles and Large Objects Developer’s Guide*.

### Table 13–10 Summary of Clob Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Clob()</code> on page 13-39</td>
<td>Clob class constructor.</td>
</tr>
<tr>
<td><code>append()</code> on page 13-39</td>
<td>Appends a <code>Clob</code> at the end of the current <code>Clob</code>.</td>
</tr>
<tr>
<td><code>close()</code> on page 13-40</td>
<td>Closes a previously opened <code>Clob</code>.</td>
</tr>
<tr>
<td><code>closeStream()</code> on page 13-40</td>
<td>Closes the <code>Stream</code> object obtained from the current <code>Clob</code>.</td>
</tr>
<tr>
<td><code>copy()</code> on page 13-40</td>
<td>Copies all or a portion of a <code>Clob</code> or <code>BFILE</code> into the current <code>Clob</code>.</td>
</tr>
<tr>
<td><code>getCharSetForm()</code> on page 13-41</td>
<td>Returns the character set form of the <code>Clob</code>.</td>
</tr>
<tr>
<td><code>getCharSetId()</code> on page 13-41</td>
<td>Returns the character set ID of the <code>Clob</code>.</td>
</tr>
<tr>
<td><code>getCharSetIdUString()</code> on page 13-41</td>
<td>Retrieves the characterset name associated with the <code>Clob</code>; <code>UString</code> version.</td>
</tr>
<tr>
<td><code>getChunkSize()</code> on page 13-41</td>
<td>Returns the smallest data size to perform efficient writes to the <code>CLOB</code>.</td>
</tr>
<tr>
<td><code>getContentType()</code> on page 13-41</td>
<td>Returns the content type of the <code>Clob</code>.</td>
</tr>
<tr>
<td><code>getOptions()</code> on page 13-42</td>
<td>Returns the <code>CLOB</code>'s <code>LobOptionValue</code> for a specified <code>LobOptionType</code>.</td>
</tr>
<tr>
<td><code>getStream()</code> on page 13-42</td>
<td>Returns data from the <code>CLOB</code> as a <code>Stream</code> object.</td>
</tr>
<tr>
<td><code>isInitialized()</code> on page 13-42</td>
<td>Tests whether the <code>Clob</code> object is initialized.</td>
</tr>
<tr>
<td><code>isNull()</code> on page 13-43</td>
<td>Tests whether the <code>Clob</code> object is atomically NULL.</td>
</tr>
</tbody>
</table>
Clob Class

**Table 13–10 (Cont.) Summary of Clob Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>isOpen() on page 13-43</td>
<td>Tests whether the Clob is open.</td>
</tr>
<tr>
<td>length() on page 13-43</td>
<td>Returns the number of characters in the current CLOB.</td>
</tr>
<tr>
<td>open() on page 13-43</td>
<td>Opens the CLOB with read or read/write access.</td>
</tr>
<tr>
<td>operator=() on page 13-43</td>
<td>Assigns a CLOB locator to the current Clob object.</td>
</tr>
<tr>
<td>operator==() on page 13-44</td>
<td>Tests whether two Clob objects are equal.</td>
</tr>
<tr>
<td>operator!=() on page 13-44</td>
<td>Tests whether two Clob objects are not equal.</td>
</tr>
<tr>
<td>read() on page 13-44</td>
<td>Reads a portion of the CLOB into a buffer.</td>
</tr>
<tr>
<td>setCharSetId() on page 13-45</td>
<td>Sets the character set ID associated with the Clob.</td>
</tr>
<tr>
<td>setCharSetIdUString() on page 13-45</td>
<td>Sets the character set ID associated with the Clob; used when the environment character set is UTF16.</td>
</tr>
<tr>
<td>setCharSetForm() on page 13-46</td>
<td>Sets the character set form associated with the Clob.</td>
</tr>
<tr>
<td>setContentType() on page 13-46</td>
<td>Sets the content type of the Clob.</td>
</tr>
<tr>
<td>setEmpty() on page 13-46</td>
<td>Sets the Clob object to empty.</td>
</tr>
<tr>
<td>setNull() on page 13-46</td>
<td>Sets the Clob object to atomically NULL.</td>
</tr>
<tr>
<td>setOptions() on page 13-47</td>
<td>Specifies a LobOptionValue for a particular LobOptionType. Enables advanced compression, encryption and deduplication of CLOBs.</td>
</tr>
<tr>
<td>trim() on page 13-47</td>
<td>Truncates the Clob to a specified length.</td>
</tr>
<tr>
<td>write() on page 13-47</td>
<td>Writes a buffer into an unopened CLOB.</td>
</tr>
<tr>
<td>writeChunk() on page 13-48</td>
<td>Writes a buffer into an open CLOB.</td>
</tr>
</tbody>
</table>

**Clob()**

Clob class constructor.

**Syntax**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clob();</td>
<td>Creates a NULL Clob object.</td>
</tr>
<tr>
<td>Clob(const Connection *connectionp);</td>
<td>Creates an uninitialized Clob object.</td>
</tr>
<tr>
<td>Clob(const Clob *srcClob);</td>
<td>Creates a copy of a Clob object.</td>
</tr>
</tbody>
</table>

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectionp</td>
<td>Connection pointer</td>
</tr>
<tr>
<td>srcClob</td>
<td>The source Clob object</td>
</tr>
</tbody>
</table>

**append()**

Appends a CLOB to the end of the current CLOB.
Syntax
void append(
    const Clob &srcClob);

Parameter Description
srcClob The CLOB to be appended to the current CLOB.

close()

Closes a CLOB.

Syntax
void close();

closeStream()

Closes the Stream object obtained from the CLOB.

Syntax
void closeStream(
    Stream *stream);

Parameter Description
stream The Stream object to be closed.

copy()

Copies a part or all of a BFILE or CLOB into the current CLOB.

OCCI does not perform any character set conversions when loading data from a Bfile into a Clob; therefore, ensure that the contents of the Bfile are character data in the server's Clob storage character set.

Syntax Description
void copy(
    const Bfile &srcBfile,
    unsigned int numBytes,
    unsigned int dstOffset = 1,
    unsigned int srcOffset = 1);

Copies a BFILE into the current CLOB.

void copy(
    const Clob &srcClob,
    unsigned int numBytes,
    unsigned int dstOffset = 1,
    unsigned int srcOffset = 1);

Copies a CLOB into the current CLOB.

If the destination CLOB has deduplication enabled, and the source and destination CLOBs are in the same column, the new CLOB is created as copy-on-write. All other settings are inherited from the source CLOB. If the destination CLOB has deduplication disabled, it is a completely new copy of the CLOB.
getCharSetForm()

Returns the character set form of the CLOB.

Syntax
CharSetForm getCharSetForm() const;

getCharSetId()

Returns the character set ID of the CLOB, in string form.

Syntax
string getCharSetId() const;

getCharSetIdUString()

Retrieves the character set name associated with the Clob; UString version.

Syntax
UString getCharSetIdUString() const;

getchunkSize()

Returns the smallest data size to perform efficient writes to the CLOB.

Syntax
unsigned int getChunkSize() const;

getContentType()

Returns the content type of the Clob. If a content type has not been assigned, returns a NULL string.
**getOptions()**

**Syntax**

```cpp
string getContentType();
```

**getOptions()**

Returns the CLOB's LobOptionValue for a specified LobOptionType.

Throws an exception if attempting to retrieve a value for an option that is not configured on the database column or partition that stores the CLOB.

**Syntax**

```cpp
LobOptionValue getOptions(
    LobOptionType optType);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optType</td>
<td>The LobOptionType setting requested. These may be combined using bitwise or (</td>
</tr>
</tbody>
</table>

**getStream()**

Returns a Stream object from the CLOB. If a stream is open, it is disallowed to open another stream on CLOB object, so the user must always close the stream before performing any Clob object operations. The client's character set id and form is used by default, unless they are explicitly set through setCharSetId() and setEmpty() calls.

**Syntax**

```cpp
Stream* getStream(
    unsigned int offset = 1,
    unsigned int amount = 0);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>The starting position at which to begin reading data from the CLOB. If offset is not specified, the data is written from the beginning of the CLOB. Valid values are numbers greater than or equal to 1.</td>
</tr>
<tr>
<td>amount</td>
<td>The total number of consecutive characters to be read. If amount is 0, the data is read from the offset value until the end of the CLOB.</td>
</tr>
</tbody>
</table>

**isInitialized()**

Tests whether the Clob object is initialized. If the Clob object is initialized, TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool isInitialized() const;
```
isNull()

Tests whether the Clob object is atomically NULL. If the Clob object is atomically NULL, TRUE is returned; otherwise, FALSE is returned.

**Syntax**
```cpp
toal isNull() const;
```

isOpen()

Tests whether the CLOB is open. If the CLOB is open, TRUE is returned; otherwise, FALSE is returned.

**Syntax**
```cpp
toal isOpen() const;
```

length()

Returns the number of characters in the CLOB.

**Syntax**
```cpp
unsigned int length() const;
```

open()

Opens the CLOB in read/write or read-only mode.

**Syntax**
```cpp
void open(
    L0bOpenMode mode = OCCI_LOB_READWRITE);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>The mode the CLOB is to be opened in. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>- OCCI_LOB_READWRITE</td>
</tr>
<tr>
<td></td>
<td>- OCCI_LOB_READONLY</td>
</tr>
</tbody>
</table>

operator=()

Assigns a CLOB to the current CLOB. The source CLOB gets copied to the destination CLOB only when the destination CLOB gets stored in the table.

**Syntax**
```cpp
Clob& operator=(
    const Clob &srcClob);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcClob</td>
<td>The Clob from which the data must be copied.</td>
</tr>
</tbody>
</table>
operator==( )

Compares two Clob objects for equality. Two Clob objects are equal if they both refer to the same CLOB. Two NULL Clob objects are not considered equal. If the Blob objects are equal, then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool operator==(  
    const Clob &srcClob) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcClob</td>
<td>The Clob object to be compared with the current Clob object.</td>
</tr>
</tbody>
</table>

operator!=( )

Compares two Clob objects for inequality. Two Clob objects are equal if they both refer to the same CLOB. Two NULL Clob objects are not considered equal. If the Clob objects are not equal, then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool operator!=(  
    const Clob &srcClob) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcClob</td>
<td>The Clob object to be compared with the current Clob object.</td>
</tr>
</tbody>
</table>

read( )

Reads a part or all of the CLOB into a buffer.

Returns the actual number of characters read for fixed-width character sets, such as UTF16, or the number of bytes read for multibyte character sets, including UTF8.

The client's character set id and form is used by default, unless they are explicitly set through setCharSetId(), setCharSetIdUString() and setCharSetForm() calls.

Note that for the second version of the method, the return value represents either the number of characters read for fixed-width character sets (UTF16), or the number of bytes read for multibyte character sets (including UTF8).

**Syntax**

```cpp
unsigned int read(  
    unsigned int amt,  
    unsigned char *buffer,  
    unsigned int bufsize,  
    unsigned int offset=1) const;
```

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads a part or all of the CLOB into a buffer.</td>
</tr>
</tbody>
</table>
**Syntax**

```cpp
unsigned int read(
    unsigned int amt,
    unsigned utext *buffer,
    unsigned int bufsize,
    unsigned int offset=1) const;
```

*Reads a part or all of the CLOB into a buffer; globalization enabled. Should be called after setting character set to OCCIUTF16 using setCharSetId() method.*

**setCharSetId()**

Sets the Character set Id associated with Clob. The character set id set is used for read/write and getStream() operations. If no value is set explicitly, the default client's character set id is used. List of character sets supported is given in Globalization Support Guide Appendix A.

**Syntax**

```cpp
void setCharSetId(
    const string &charset);
```

**setCharSetIdUString()**

Sets the character set id associated with Clob; used when the environment's character set is UTF16. The charset id set is used for read, write and getStream() operations.

**Syntax**

```cpp
void setCharSetIdUString(
    const string &charset);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charset</td>
<td>Oracle supported character set name, such as E8DEC, ZHT16BIG5, or OCCIUTF16.</td>
</tr>
</tbody>
</table>
**setCharSetForm()**

Sets the character set form associated with the CLOB. The charset form set is used for read, write and `getStream()` operations. If no value is set explicitly, by default, `OCCI_SQLCS_IMPLICIT` is used.

**Syntax**

```cpp
void setCharSetForm(
    CharSetForm csfrm );
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csfrm</td>
<td>The charset form for Clob.</td>
</tr>
</tbody>
</table>

**setContentType()**

Sets the content type of the Clob. If the Clob is not a SecureFile, throws an exception.

**Syntax**

```cpp
void setContentType(
    const string contenttype);`
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contenttype</td>
<td>The content type of the Clob; an ASCII Mime compliant string.</td>
</tr>
</tbody>
</table>

**setEmpty()**

Sets the Clob object to empty.

**Syntax**

```cpp
void setEmpty();
void setEmpty(
    const Connection* connectionp);`n```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectionp</td>
<td>The new connection pointer for the Clob object.</td>
</tr>
</tbody>
</table>

**setNull()**

Sets the Clob object to atomically NULL.

**Syntax**

```cpp
void setNull();
```
setOptions()

Specifies a LobOptionValue for a particular LobOptionType. Enables advanced compression, encryption and deduplication of CLOBs. See Table 7–1, "Values of Type LobOptionType" and Table 7–2, "Values of Type LobOptionValue".

Throws an exception if attempting to set or un-set an option that is not configured on the database column or partition that stores the CLOB.

Throws an exception if attempting to turn off encryption in an encrypted CLOB column.

**Syntax**

```c
void setOptions(
    LobOptionType optType,
    LobOptionValue optValue);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optType</td>
<td>The LobOptionType setting being specified. These may be combined using bitwise or (</td>
</tr>
<tr>
<td>optValue</td>
<td>The LobOptionValue setting for the LobOptionType specified by the optType parameter</td>
</tr>
</tbody>
</table>

**trim()**

Truncates the CLOB to the new length specified.

**Syntax**

```c
void trim(
    unsigned int newlen);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newlen</td>
<td>The new length of the CLOB. Valid values are numbers less than or equal to the current length of the CLOB.</td>
</tr>
</tbody>
</table>

**write()**

Writes data from a buffer into a CLOB.

This method implicitly opens the CLOB, copies the buffer into the CLOB, and implicitly closes the CLOB. If the CLOB is open, use writeChunk() instead. The actual number of characters written is returned. The client’s character set id and form is used by default, unless they are explicitly set through setCharSetId() and setCharSetForm() calls.

**Syntax**

```c
unsigned int write(
    unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset=1);
```

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>Writes data from a buffer into a CLOB.</td>
</tr>
</tbody>
</table>
writeChunk() (1 of 2)

**Syntax**

```c
unsigned int write(
    unsigned int amt,
    utext *buffer,
    unsigned int bufsize,
    unsigned int offset=1);
```

**Description**

Writes data from a buffer into a previously opened CLOB. Returns the actual number of characters. The client's character set id and form is used by default, unless they are explicitly set through `setCharSetId()` and `setCharSetForm()` calls.

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>amt</code></td>
<td>The amount parameter represents:</td>
</tr>
<tr>
<td></td>
<td>- number of characters written for fixed-width character sets (UTF16)</td>
</tr>
<tr>
<td></td>
<td>- number of bytes written for multibyte character sets (including UTF8)</td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>The buffer containing the data to be written to the CLOB.</td>
</tr>
<tr>
<td><code>buffsize</code></td>
<td>The size of the buffer containing the data to be written to the CLOB. Valid values are numbers greater than or equal to <code>amt</code>.</td>
</tr>
<tr>
<td><code>offset</code></td>
<td>The starting position at which to begin writing data into the CLOB. If <code>offset</code> is not specified, the data is written from the beginning of the CLOB. Valid values are numbers greater than or equal to 1.</td>
</tr>
</tbody>
</table>

writeChunk() (2 of 2)

**Syntax**

```c
unsigned int writeChunk(
    unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset=1);
```

**Description**

Writes data from a buffer into a previously opened CLOB.

```c
unsigned int writeChunk(
    unsigned int amt,
    utext *buffer,
    unsigned int bufsize,
    unsigned int offset=1);
```

**Description**

Writes data from a UTF16 buffer into a CLOB; globalization enabled. Should be called after setting character set to OCCIUTF16 using `setCharSetIdUString()` method.

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>amt</code></td>
<td>The amount parameter represents either a number of characters written for fixed-width character sets (UTF16) or a number of bytes written for multibyte character sets (including UTF8)</td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>The buffer containing the data to be written to the CLOB.</td>
</tr>
<tr>
<td><code>buffsize</code></td>
<td>The size of the buffer containing the data to be written to the CLOB. Valid values are numbers greater than or equal to <code>amt</code>.</td>
</tr>
<tr>
<td><code>offset</code></td>
<td>The starting position at which to begin writing data into the CLOB. If <code>offset</code> is not specified, the data is written from the beginning of the CLOB. Valid values are numbers greater than or equal to 1.</td>
</tr>
</tbody>
</table>
Connection Class

The Connection class represents a connection with a specific database. Inside the connection, SQL statements are executed and results are returned.

Table 13–11  Enumerated Values Used by Connection Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>FailOverEventType</td>
<td>■ FO_BEGIN indicates that a lost connection has been detected; failover is starting.</td>
</tr>
<tr>
<td></td>
<td>■ FO_END indicates that a failover completed successfully; the Connection is ready for use.</td>
</tr>
<tr>
<td></td>
<td>■ FO_ABORT indicates that the failover was unsuccessful; it is not be attempted again.</td>
</tr>
<tr>
<td></td>
<td>■ FO_REAUTH indicates that the user session has been reauthenticated.</td>
</tr>
<tr>
<td></td>
<td>■ FO_ERROR indicates that a failover was unsuccessful; the application can handle the error and retry failover.</td>
</tr>
<tr>
<td>FailOverType</td>
<td>■ FO_NONE indicates that the user requested no protection for failover.</td>
</tr>
<tr>
<td></td>
<td>■ FO_SESSION indicates that the user requested only session failover.</td>
</tr>
<tr>
<td></td>
<td>■ FO_SELECT indicates that the use requested select failover.</td>
</tr>
<tr>
<td>ProxyType</td>
<td>■ PROXY_DEFAULT is the database user name.</td>
</tr>
</tbody>
</table>

Table 13–12  Summary of Connection Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>changePassword() on 13-50</td>
<td>Changes the password for the current user.</td>
</tr>
<tr>
<td>commit() on page 13-51</td>
<td>Commits changes made since the previous commit or rollback and release any database locks held by the session.</td>
</tr>
<tr>
<td>createStatement() on page 13-51</td>
<td>Creates a Statement object to execute SQL statements.</td>
</tr>
<tr>
<td>flushCache() on page 13-52</td>
<td>Flushes the object cache associated with the connection.</td>
</tr>
<tr>
<td>getClientCharSet() on page 13-52</td>
<td>Returns the default client character set.</td>
</tr>
<tr>
<td>getClientNCHARCharSet() on page 13-52</td>
<td>Returns the default client NCHAR character set.</td>
</tr>
<tr>
<td>getClientVersion() on page 13-52</td>
<td>Returns the version of the client used.</td>
</tr>
<tr>
<td>getMetaData() on page 13-53</td>
<td>Returns the metadata for an object accessible from the connection.</td>
</tr>
<tr>
<td>getOCIServer() on page 13-53</td>
<td>Returns the OCI server context associated with the connection.</td>
</tr>
<tr>
<td>getOCIServiceContext() on page 13-54</td>
<td>Returns the OCI service context associated with the connection.</td>
</tr>
</tbody>
</table>
Table 13–12  (Cont.) Summary of Connection Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>getOClISession() on page 13-54</td>
<td>Returns the OCI session context associated with the connection.</td>
</tr>
<tr>
<td>getServerVersion() on page 13-54</td>
<td>Returns the version of the Oracle server used, as string.</td>
</tr>
<tr>
<td>getServerVersionUString() on page 13-54</td>
<td>Returns the version of the Oracle server used, as a UString.</td>
</tr>
<tr>
<td>getStmtCacheSize() on page 13-54</td>
<td>Retrieves the size of the statement cache.</td>
</tr>
<tr>
<td>getTag() on page 13-54</td>
<td>Returns the tag associated with the connection.</td>
</tr>
<tr>
<td>isCached() on page 13-55</td>
<td>Determines if the specified statement is cached.</td>
</tr>
<tr>
<td>pinVectorOfRefs() on page 13-55</td>
<td>Pins an entire vector of Ref objects into object cache in a single round trip; pinned objects are available through an OUT parameter vector.</td>
</tr>
<tr>
<td>postToSubscriptions() on page 13-56</td>
<td>Posts notifications to subscriptions.</td>
</tr>
<tr>
<td>readVectorOfBfiles() on page 13-56</td>
<td>Reads multiple Bfiles in a single server round-trip.</td>
</tr>
<tr>
<td>readVectorOfBlobs() on page 13-56</td>
<td>Reads multiple Blobs in a single server round-trip.</td>
</tr>
<tr>
<td>readVectorOfClobs() on page 13-57</td>
<td>Reads multiple Clos in a single server round-trip.</td>
</tr>
<tr>
<td>registerSubscriptions() on page 13-58</td>
<td>Registers several Subscriptions for notification.</td>
</tr>
<tr>
<td>rollback() on page 13-58</td>
<td>Rolls back all changes made since the previous commit or rollback and release any database locks held by the session.</td>
</tr>
<tr>
<td>setStmtCacheSize() on page 13-58</td>
<td>Enables or disables statement caching.</td>
</tr>
<tr>
<td>setTAFNotify() on page 13-59</td>
<td>Registers failover callback function on the Connection object.</td>
</tr>
<tr>
<td>terminateStatement() on page 13-59</td>
<td>Closes a Statement object and free all resources associated with it.</td>
</tr>
<tr>
<td>unregisterSubscription() on page 13-60</td>
<td>Unregisters a Subscription, turns off its notifications.</td>
</tr>
<tr>
<td>writeVectorOfBlobs() on page 13-60</td>
<td>Writes multiple Blobs in a single server round-trip.</td>
</tr>
<tr>
<td>writeVectorOfClobs() on page 13-60</td>
<td>Writes multiple Clos in a single server round-trip.</td>
</tr>
</tbody>
</table>

changePassword()

Changes the password of the user currently connected to the database.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void changePassword(</td>
<td>Changes the password of the user.</td>
</tr>
<tr>
<td>const string &amp;user,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;oldPassword,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;newPassword) = 0;</td>
<td></td>
</tr>
<tr>
<td>void changePassword()</td>
<td>Changes the password of the user (Unicode support).</td>
</tr>
<tr>
<td>const UString &amp;user,</td>
<td>The client Environment should be initialized in OCCIUTIF16 mode.</td>
</tr>
<tr>
<td>const UString &amp;oldPassword,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;newPassword) = 0;</td>
<td></td>
</tr>
</tbody>
</table>
commit()

Commits all changes made since the previous commit or rollback, and releases any database locks currently held by the session.

Syntax

```cpp
void commit()=0;
```

createStatement()

Creates a `Statement` object with the SQL statement specified.

Note that for the caching-enabled version of this method, the cache is initially searched for a statement with a matching `tag`, which is returned. If no match is found, the cache is searched again for a statement that matches the `sql` parameter, which is returned. If no match is found, a new statement with a `NULL tag` is created and returned. If the `sql` parameter is empty and the `tag` search fails, this call generates an `ERROR`.

Also note that non-caching versions of this method always create and return a new statement.

Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| `Statement* createStatement(
   const string &sql="")=0;` | Searches the cache for a specified SQL statement and returns it; if not found, creates a new statement. |
| `Statement* createStatement(
   const string &sql,
   const string &tag)=0;` | Searches the cache for a statement with a matching `tag`; if not found, creates a new statement with the specified SQL content. |
| `Statement* createStatement(
   const UString &sql)=0;` | Searches the cache for a specified SQL statement and returns it; if not found, creates a new statement. Globalization enabled. |
| `Statement* createStatement(
   const Ustring &sql,
   const Ustring &tag)=0;` | Searches the cache for a matching `tag` and returns it; if not found, creates a new statement with the specified SQL content. Globalization enabled. |

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sql</code></td>
<td>The SQL string to be associated with the statement object.</td>
</tr>
<tr>
<td><code>tag</code></td>
<td>The tag whose associated statement must be retrieved from the cache. Ignored if statement caching is disabled.</td>
</tr>
</tbody>
</table>
flushCache()

Flushes the object cache associated with the connection.

Syntax
void flushCache()=0;

cgetClientCharSet()

Returns the session’s character set.

Syntax
string getClientCharSet() const=0;

cgetClientCharSetUString()

Returns the globalization enabled client character set in UString.

Syntax
UString getClientCharSetUString() const=0;

cgetClientNCHARCharSet()

Returns the session’s NCHAR character set.

Syntax
string getClientNCHARCharSet() const=0;

cgetClientNCHARCharSetUString()

Returns the globalization enabled client NCHAR character set in UString.

Syntax
UString getClientNCHARCharSetUString() const=0;

cgetClientVersion()

Returns the version of the client library the application is using at run time.
This is used by applications to determine the version of the OCCI client at run time, and if the application uses several separate code paths that use several different client patchsets.

The values of parameters majorVersion and minorVersion use macros OCCI_MAJOR_VERSION and OCCI_MINOR_VERSION, respectively. These macros define the major and minor versions of the OCCI client library. Compares the versions returned.
Syntax

```c
void getClientVersion(
    int &majorVersion,
    int &minorVersion,
    int &updateNum,
    int &patchNumber,
    int &portUpdateNum)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>majorVersion</td>
<td>The major version of the client library.</td>
</tr>
<tr>
<td>minorVersion</td>
<td>The minor version of the client library.</td>
</tr>
<tr>
<td>updateNum</td>
<td>The update number.</td>
</tr>
<tr>
<td>patchNumber</td>
<td>The number of the patch applied to the library.</td>
</tr>
<tr>
<td>portUpdateNum</td>
<td>The number of the port-specific port update applied to the library.</td>
</tr>
</tbody>
</table>

**getMetaData()**

Returns metadata for an object in the database.

Syntax

```c
MetaData getMetaData(
    const string &object,
    MetaData::ParamType prmtyp=MetaData::PTYPE_UNK) const=0;

MetaData getMetaData(
    const UString &object,
    MetaData::ParamType prmtyp=MetaData::PTYPE_UNK) const=0;

MetaData getMetaData(
    const RefAny &ref) const=0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>The SQL string to be associated with the statement object.</td>
</tr>
<tr>
<td>prmtyp</td>
<td>The type of the schema object being described, as defined by the enumerated ParamType of the MetaData class, Table 13-27 on page 13-128</td>
</tr>
<tr>
<td>ref</td>
<td>A REF to the Type Descriptor Object (TDO) of the type to be described.</td>
</tr>
</tbody>
</table>

**getOCIServer()**

Returns the OCI server context associated with the connection.

Syntax

```c
OCIServer* getOCIServer() const=0;
```
getOCIServiceContext()

Returns the OCI service context associated with the connection.

Syntax
OCISvcCtx* getOCIServiceContext() const=0;

getOCIErrorException()

Returns the OCI error context associated with the connection.

Syntax
OCIErrorException* getOCIErrorException() const=0;

getServerVersion()

Returns the version of the database server, as a string, used by the current Connection object. This can be used when an application uses several separate code paths and connects to several different server versions.

Syntax
string getServerVersion() const;

getServerVersionUString()

Returns the version of the database server, as a UString, used by the current Connection object. This can be used when an application uses several separate code paths and connects to several different server versions.

Syntax
UString getServerVersionUString() const;

getStmtCacheSize()

Retrieves the size of the statement cache.

Syntax
unsigned int getStmtCacheSize() const=0;

g.getTag()

Returns the tag associated with the connection. Valid only for connections from a stateless connection pool.

Syntax
string getTag() const=0;
isCached()

Determines if the specified statement is cached.

```
bool isCached(
    const string &sql,
    const string &tag="")=0;
```

Searches the cache for a statement with a matching tag. If the tag is not specified, the cache is searched for a matching SQL statement.

```
bool isCached(
    const Ustring &sql,
    const Ustring &tag)=0;
```

Searches the cache for a statement with a matching tag. If the tag is not specified, the cache is searched for a matching SQL statement. Globalization enabled.

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td>The SQL string to be associated with the statement object.</td>
</tr>
<tr>
<td>tag</td>
<td>The tag whose associated statement must be retrieved from the cache. Ignored if statement caching is disabled.</td>
</tr>
</tbody>
</table>

pinVectorOfRefs()

Pins an entire vector of `Ref` objects into object cache in a single round-trip. Pinned objects are available through an OUT parameter vector.

```
template <class T> void
pinVectorOfRefs(
    const Connection *conn,
    vector <Ref<T>> & vect,
    vector <T*> & vectObj,
    LockOptions lockOpt=OCCI_LOCK_NONE);
```

Returns the objects.

```
template <class T> void
pinVectorOfRefs(
    const Connection *conn,
    vector <Ref<T>> & vect,
    LockOptions lockOpt=OCCI_LOCK_NONE);
```

Does not explicitly return the objects; an application must dereference a particular `Ref` object by a `ptr()` call, which returns a previously pinned object.

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>Connection</td>
</tr>
<tr>
<td>vect</td>
<td>Vector of <code>Ref</code> objects that are pinned.</td>
</tr>
<tr>
<td>vectObj</td>
<td>Vector that contains objects after the pinning operation is complete; an OUT parameter.</td>
</tr>
<tr>
<td>lockOpt</td>
<td>Lock option used during the pinning of the array, as defined by <code>LockOptions</code> in Table 13–2 on page 13-6. The only supported value is <code>OCCI_LOCK_NONE</code>.</td>
</tr>
</tbody>
</table>
**postToSubscriptions()**

Posts notifications to subscriptions.

The Subscription object must have a valid subscription name, and the namespace should be set to NS_ANONYMOUS. The payload must be set before invoking this call; otherwise, the payload is assumed to be NULL and is not delivered.

The caller has to preserve the payload until the posting call is complete. This call provides a best-effort guarantee; a notification is sent to registered clients at most once.

This call is primarily used for light-weight notification and is useful for dealing with several system events. If the application needs more rigid guarantees, it can use the Oracle Streams Advanced Queuing functionality.

**Syntax**

```cpp
void postToSubscriptions(
    const vector<aq::Subscription>& sub);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>The vector of subscriptions that receive postings.</td>
</tr>
</tbody>
</table>

**readVectorOfBfiles()**

Reads multiple Bfiles in a single server round-trip. All Bfiles must be open for reading.

**Syntax**

```cpp
void readVectorOfBfiles(
    const Connection *conn,
    vector<Bfile> &vec,
    oraub8 *byteAmts,
    oraub8 *offsets,
    unsigned char *buffers[],
    oraub8 *bufferLengths);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>Connection.</td>
</tr>
<tr>
<td>vec</td>
<td>Vector of Bfile objects; each Bfile must be open for reading.</td>
</tr>
<tr>
<td>byteAmts</td>
<td>Array of amount of bytes to read from the individual Bfiles. The actual number of bytes read from each Bfile is returned in this array.</td>
</tr>
<tr>
<td>offsets</td>
<td>Array of offsets, starting position where reading from the Bfiles starts.</td>
</tr>
<tr>
<td>buffers</td>
<td>Array of pointers to buffers into which the data is read.</td>
</tr>
<tr>
<td>bufferLengths</td>
<td>Array of sizes of each buffer, in bytes.</td>
</tr>
</tbody>
</table>

**readVectorOfBlobs()**

Reads multiple BLOBs in a single server round-trip.
Syntax

```c
void readVectorOfBlobs(
    const Connection *conn,
    vector<Blob> &vec,
    oraub8 *byteAmts,
    oraub8 *offsets,
    unsigned char *buffers[],
    oraub8 *bufferLengths);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>Connection.</td>
</tr>
<tr>
<td>vec</td>
<td>Vector of Blob objects.</td>
</tr>
<tr>
<td>byteAmts</td>
<td>Array of amount of bytes to read from the individual Blobs. The actual number of bytes read from each Blob is returned in this array.</td>
</tr>
<tr>
<td>offsets</td>
<td>Array of offsets, starting position where reading from the Blobs starts.</td>
</tr>
<tr>
<td>buffers</td>
<td>Array of pointers to buffers into which the data is read.</td>
</tr>
<tr>
<td>bufferLengths</td>
<td>Array of sizes of each buffer, in bytes.</td>
</tr>
</tbody>
</table>

**readVectorOfClobs()**

Reads multiple Clobs in a single server round-trip. All Clobs should be in the same characterset form and belong to the same characterset ID.

Syntax

```c
void readVectorOfClobs(
    const Connection *conn,
    vector<Clob> &vec,
    oraub8 *byteAmts,
    araub8 *charAmts,
    oraub8 *offsets,
    utext *buffers[],
    oraub8 *bufferLengths);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>Connection.</td>
</tr>
<tr>
<td>vec</td>
<td>Vector of Clob objects.</td>
</tr>
<tr>
<td>byteAmts</td>
<td>Array of amount of bytes to read from the individual Clobs. Only used if the charAmts is NULL, or 0 for any Clob index. Returns the number of bytes read for each Clob.</td>
</tr>
</tbody>
</table>
registerSubscriptions()

Registers subscriptions for notification.

New client processes and existing processes that restart after a shut down must register for all subscriptions of interest. If the client stays up during a server shut down and restart, this client continues to receive notifications for DISCONNECTED registrations, but not for CONNECTED registrations because they are lost during the server down time.

Syntax

```cpp
void registerSubscriptions(
    const vector<aq::Subscription>& sub)=0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>Vector of subscriptions that are registered for notification.</td>
</tr>
</tbody>
</table>

rollback()

Drops all changes made since the previous commit or rollback, and releases any database locks currently held by the session.

Syntax

```cpp
void rollback()=0;
```

setStmtCacheSize()

Enables or disables statement caching. A nonzero value enables statement caching, with a cache of specified size. A zero value disables caching.

Syntax

```cpp
void setStmtCacheSize(
    unsigned int cacheSize)=0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacheSize</td>
<td>The maximum number of statements in the cache.</td>
</tr>
</tbody>
</table>
**setTAFNotify()**

Registers the failover callback function on the Connection object for which failover is configured and must be detected.

The failover callback should return OCCI_SUCCESS to indicate that OCCI can continue with default processing. The failover event, foEvent, is defined in Table 13–11 on page 13-49. When the foEvent is FO_ERROR, the callback function may return either FO_RETRY to indicate that failover must be attempted again, or OCCI_SUCCESS to end failover attempts.

**Syntax**

```c
void setTAFNotify(
    int (*notifyFn)(
        Environment *env,
        Connection *conn,
        void *ctx,
        FailOverType foType,
        FailOverEventType foEvent,
        void *ctxTAF)
)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>notifyFn</td>
<td>The user defined callback function invoked during failover events.</td>
</tr>
<tr>
<td>env</td>
<td>Environment object from which the failing Connection was created.</td>
</tr>
<tr>
<td>conn</td>
<td>The failing Connection on which the callback function is registered.</td>
</tr>
<tr>
<td>ctx</td>
<td>Context supplied by the user when registering the callback.</td>
</tr>
<tr>
<td>foType</td>
<td>The configured FailOverType, values FO_SESSION or FO_SELECT, as defined in Table 13–11 on page 13-49.</td>
</tr>
<tr>
<td>foEvent</td>
<td>Failover event type that is triggering the callback; the FailOverEventType, values FO_BEGIN, FO_END, FO_ABORT and FO_ERROR as defined in Table 13–11 on page 13-49.</td>
</tr>
<tr>
<td>ctxTAF</td>
<td>User context passed back to the callback function at invocation.</td>
</tr>
</tbody>
</table>

**terminateStatement()**

Closes a Statement object.

**Syntax**

```c
void terminateStatement(
    Statement *stmt)=0;
void terminateStatement(
    Statement *stmt,
    const string &tag)=0;
void terminateStatement(
    Statement* stmt,
    const UString &tag) = 0;
```

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void terminateStatement({</td>
<td>Closes a Statement object and frees all resources associated with it.</td>
</tr>
<tr>
<td>Statement *stmt}=0;</td>
<td></td>
</tr>
<tr>
<td>void terminateStatement({</td>
<td>Releases statement back to the cache after adding an optional tag, a string.</td>
</tr>
<tr>
<td>Statement *stmt,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;tag)=0;</td>
<td></td>
</tr>
<tr>
<td>void terminateStatement({</td>
<td>Releases statement back to the cache after adding an optional tag, a UString.</td>
</tr>
<tr>
<td>Statement* stmt,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;tag) = 0;</td>
<td></td>
</tr>
</tbody>
</table>
**unregisterSubscription()**

Unregisters a Subscription, turning off its notifications.

**Syntax**

```cpp
void unregisterSubscription(
    const aq::Subscription& sub)=0;
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>Subscription whose notifications is turned off.</td>
</tr>
</tbody>
</table>

**writeVectorOfBlobs()**

Writes multiple Blobs in a single server round-trip.

**Syntax**

```cpp
void writeVectorOfBlobs(
    const Connection *conn,
    vector<Blob> &vec,
    oraub8 *byteAms,  
    oraub8 *offsets,  
    unsigned char *buffers[], 
    oraub8 *bufferLengths);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>Connection.</td>
</tr>
<tr>
<td>vec</td>
<td>Vector of Blob objects.</td>
</tr>
<tr>
<td>byteAms</td>
<td>Array of amount of bytes to write to the individual Blobs.</td>
</tr>
<tr>
<td>offsets</td>
<td>Array of offsets, starting position where writing to the Blobs starts.</td>
</tr>
<tr>
<td>buffers</td>
<td>Array of pointers to buffers from which the data is written.</td>
</tr>
<tr>
<td>bufferLengths</td>
<td>Array of sizes of each buffer, in bytes.</td>
</tr>
</tbody>
</table>

**writeVectorOfClobs()**

Writes multiple Clobs in a single server round-trip. All Clobs should be in the same characterset form and belong to the same characterset ID.
Syntax Description

void writeVectorOfClobs(
   const Connection *conn,
   vector<Clob> &vec,
   oraub8 *byteAmts,
   araub8 *charAmts,
   oraub8 *offsets,
   unsigned char *buffers[],
   oraub8 *bufferLengths);

void writeVectorOfClobs(
   const Connection *conn,
   vector<Clob> &vec,
   oraub8 *byteAmts,
   araub8 *charAmts,
   oraub8 *offsets,
   utext *buffers[],
   oraub8 *bufferLengths);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>Connection.</td>
</tr>
<tr>
<td>vec</td>
<td>Vector of Clob objects.</td>
</tr>
<tr>
<td>byteAmts</td>
<td>Array of amount of bytes to write to the individual Clobs. Only used if the charAmts is NULL or 0 for any Clob index. Returns the number of bytes written for each Clob.</td>
</tr>
<tr>
<td>charAmts</td>
<td>Array of amount of characters to write to individual Clobs. Returns the number of characters read for each Clob.</td>
</tr>
<tr>
<td>offsets</td>
<td>Array of offsets, starting position where writing to the Clobs starts, in characters.</td>
</tr>
<tr>
<td>buffers</td>
<td>Array of pointers to buffers from which the data is written.</td>
</tr>
<tr>
<td>bufferLengths</td>
<td>Array of sizes of each buffer, in bytes.</td>
</tr>
</tbody>
</table>
ConnectionPool Class

The ConnectionPool class represents a pool of connections for a specific database.

### Table 13–13 Summary of ConnectionPool Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>createConnection() on page 13-62</td>
<td>Creates a pooled connection.</td>
</tr>
<tr>
<td>createProxyConnection() on page 13-63</td>
<td>Creates a proxy connection.</td>
</tr>
<tr>
<td>getBusyConnections() on page 13-63</td>
<td>Returns the number of busy connections in the connection pool.</td>
</tr>
<tr>
<td>getIncrConnections() on page 13-64</td>
<td>Returns the number of incremental connections in the connection pool.</td>
</tr>
<tr>
<td>getMaxConnections() on page 13-64</td>
<td>Returns the maximum number of connections in the connection pool.</td>
</tr>
<tr>
<td>getMinConnections() on page 13-64</td>
<td>Returns the minimum number of connections in the connection pool.</td>
</tr>
<tr>
<td>getOpenConnections() on page 13-64</td>
<td>Returns the number of open connections in the connection pool.</td>
</tr>
<tr>
<td>getName() on page 13-64</td>
<td>Returns the name of the connection pool.</td>
</tr>
<tr>
<td>getStmtCacheSize() on page 13-64</td>
<td>Retrieves the size of the statement cache.</td>
</tr>
<tr>
<td>getTimeOut() on page 13-65</td>
<td>Returns the time out period for a connection in the connection pool.</td>
</tr>
<tr>
<td>setErrorOnBusy() on page 13-65</td>
<td>Specifies that a SQLException should be generated when all connections in the connection pool are busy and no further connections can be opened.</td>
</tr>
<tr>
<td>setPoolSize() on page 13-65</td>
<td>Sets the minimum, maximum, and incremental number of pooled connections for the connection pool.</td>
</tr>
<tr>
<td>setStmtCacheSize() on page 13-65</td>
<td>Enables or disables statement caching.</td>
</tr>
<tr>
<td>setTimeout() on page 13-65</td>
<td>Sets the time out period, in seconds, for a connection in the connection pool.</td>
</tr>
<tr>
<td>terminateConnection() on page 13-66</td>
<td>Destroys the connection.</td>
</tr>
</tbody>
</table>

### createConnection()

Creates a pooled connection.

#### Syntax

- **Syntax**

  ```cpp
  Connection* createConnection(
    const string &userName,
    const string &password)=0;
  ```
  
  Creates a pooled connection. If the `userName` and `password` are both `NULL`, the connection is externally authenticated.

- **Syntax**

  ```cpp
  Connection* createConnection(
    const UString &username,
    const UString &password)=0;
  ```
  
  Creates a globalization enabled pooled connection.
createProxyConnection()

Creates a proxy connection from the connection pool.

Syntax

```cpp
Connection* createProxyConnection(
    const string &name,
    Connection::ProxyType
    proxyType=Connection::PROXY_DEFAULT)=0;
```

`createProxyConnection()` creates a proxy connection.

```cpp
Connection* createProxyConnection(
    const UString &name,
    Connection::ProxyType
    proxyType=Connection::PROXY_DEFAULT)=0;
```

`createProxyConnection()` creates a globalization enabled proxy connection.

```cpp
Connection* createProxyConnection(
    const string &name,
    string roles[],
    int numRoles,
    Connection::ProxyType
    proxyType=Connection::PROXY_DEFAULT)=0;
```

`createProxyConnection()` creates a proxy connection for several roles.

```cpp
Connection* createProxyConnection(
    const UString &name,
    string roles[],
    string roles[],
    unsigned int numRoles,
    Connection::ProxyType
    proxyType=Connection::PROXY_DEFAULT)=0;
```

`createProxyConnection()` creates a globalization enabled proxy connection for several roles.

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The user name to connect with.</td>
</tr>
<tr>
<td>roles</td>
<td>The roles to activate on the database server.</td>
</tr>
<tr>
<td>numRoles</td>
<td>The number of roles to activate on the database server.</td>
</tr>
<tr>
<td>proxyType</td>
<td>The type of proxy authentication to perform, ProxyType, defined in Table 13-11 on page 13-49. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>■ PROXY_DEFAULT representing a database user name.</td>
</tr>
</tbody>
</table>

getBusyConnections()

Returns the number of busy connections in the connection pool. When using database resident connection pooling, this is the number of persistent connections to the Connection Broker.

Syntax

```cpp
unsigned int getBusyConnections() const=0;
```
getIncrConnections()

Returns the number of incremental connections in the connection pool. When using database resident connection pooling, this is the number of persistent connections to the Connection Broker.

**Syntax**

```cpp
unsigned int getIncrConnections() const=0;
```

getMaxConnections()

Returns the maximum number of connections in the connection pool. When using database resident connection pooling, this is the number of persistent connections to the Connection Broker.

**Syntax**

```cpp
unsigned int getMaxConnections() const=0;
```

getMinConnections()

Returns the minimum number of connections in the connection pool. When using database resident connection pooling, this is the number of persistent connections to the Connection Broker.

**Syntax**

```cpp
unsigned int getMinConnections() const=0;
```

getOpenConnections()

Returns the number of open connections in the connection pool. When using database resident connection pooling, this is the number of persistent connections to the Connection Broker.

**Syntax**

```cpp
unsigned int getOpenConnections() const=0;
```

getPoolName()

Returns the name of the connection pool.

**Syntax**

```cpp
string getPoolName() const=0;
```

getStmtCacheSize()

Retrieves the size of the statement cache.
**SqlConnection Class**

getStmtCacheSize() const=0;

**getTimeOut()**

Returns the time out period of a connection in the connection pool.

**Syntax**

unsigned int getTimeOut() const=0;

setErrorOnBusy()

Specifies that a SQLException is to be generated when all connections in the connection pool are busy and no further connections can be opened.

**Syntax**

void setErrorOnBusy()=0;

setPoolSize()

Sets the minimum, maximum, and incremental number of pooled connections for the connection pool.

**Syntax**

void setPoolSize(
    unsigned int minConn = 0,
    unsigned int maxConn = 1,
    unsigned int incrConn = 1)=0;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minConn</td>
<td>The minimum number of connections for the connection pool.</td>
</tr>
<tr>
<td>maxConn</td>
<td>The maximum number of connections for the connection pool.</td>
</tr>
<tr>
<td>incrConn</td>
<td>The incremental number of connections for the connection pool.</td>
</tr>
</tbody>
</table>

setStmtCacheSize()

Enables or disables statement caching. A nonzero value enables statement caching, with a cache of specified size. A zero value disables caching.

**Syntax**

void setStmtCacheSize(
    unsigned int cacheSize)=0;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacheSize</td>
<td>The size of the statement cache.</td>
</tr>
</tbody>
</table>
setTimeOut()

Sets the time out period for a connection in the connection pool. OCCI terminates any connections related to this connection pool that have been idle for longer than the time out period specified.

If this attribute is not set, the least recently used sessions are timed out when pool space is required. Oracle only checks for timed out sessions when it releases a session back to the pool.

**Syntax**

```c
void setTimeOut(
    unsigned int connTimeOut = 0)=0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connTimeOut</td>
<td>The timeout period in number of seconds.</td>
</tr>
</tbody>
</table>

**terminateConnection()**

Terminates the pooled connection or proxy connection.

**Syntax**

```c
void terminateConnection(
    Connection *connection)=0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>The pooled connection or proxy connection to terminate.</td>
</tr>
</tbody>
</table>
Consumer Class

The Consumer class supports dequeuing of Messages and controls the dequeuing options.

Table 13–14 Enumerated Values Used by Consumer Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>DequeMode</td>
<td>- DEQ_BROWSE indicates that the message should be read without acquiring a lock; equivalent to a SELECT.</td>
</tr>
<tr>
<td></td>
<td>- DEQ_LOCKED indicates that the message should be read. Get its write lock, which lasts s for the duration of the transaction; equivalent to a SELECT FOR UPDATE.</td>
</tr>
<tr>
<td></td>
<td>- DEQ_REMOVE indicates that the message should be read. Update or delete it; the message can be retained in the queue table based on the retention properties. This is the default setting.</td>
</tr>
<tr>
<td></td>
<td>- DEQ_REMOVE_NODATA indicates that the receipt of the message should be confirmed, but its actual content should not be delivered.</td>
</tr>
<tr>
<td>Navigation</td>
<td>- DEQ_FIRST_MSG indicates that the first available message on the queue that matches the search criteria must be retrieved. Resets the position to the beginning of the queue.</td>
</tr>
<tr>
<td></td>
<td>- DEQ_NEXT_TRANSACTION indicates that the next available message on the queue that matches the search criteria must be retrieved. If the previous message belongs to a message group, AQ retrieves the next available message that matches the search criteria and belongs to the message group. This is the default setting.</td>
</tr>
<tr>
<td></td>
<td>- DEQ_NEXT_MSG indicates that the remainder of the current transaction group, if any, should be skipped. The first message of the next transaction group may then be retrieved. This option can only be used if message grouping is enabled for the current queue.</td>
</tr>
<tr>
<td>Visibility</td>
<td>- DEQ_IMMEDIATE indicates that the dequeued message is not part of the current transaction. It constitutes a transaction on its own.</td>
</tr>
<tr>
<td></td>
<td>- DEQ_ON_COMMIT indicates that the dequeue is part of the current transaction. This is the default setting.</td>
</tr>
<tr>
<td>DequeWaitOption</td>
<td>- DEQ_WAIT_FOREVER indicates that the consumer waits for the Message indefinitely.</td>
</tr>
<tr>
<td></td>
<td>- DEQ_NO_WAIT indicates that there should be no wait if there are no messages on the queue.</td>
</tr>
</tbody>
</table>

Table 13–15 Summary of Consumer Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer() on page 13-68</td>
<td>Consumer class constructor.</td>
</tr>
<tr>
<td>getConsumerName() on page 13-69</td>
<td>Retrieves the name of the Consumer.</td>
</tr>
<tr>
<td>getCorrelationId() on page 13-69</td>
<td>Retrieves the correlation id of the message that is to be dequeued.</td>
</tr>
<tr>
<td>getDequeueMode() on page 13-69</td>
<td>Retrieves the dequeue mode of the Consumer.</td>
</tr>
<tr>
<td>getMessageIdToDequeue() on page 13-69</td>
<td>Retrieves the id of the message that is dequeued.</td>
</tr>
<tr>
<td>getQueueName() on page 13-70</td>
<td>Gets the name of the queue used by the consumer.</td>
</tr>
</tbody>
</table>
### Table 13–15 (Cont.) Summary of Consumer Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getPositionOfMessage() on page 13-69</td>
<td>Retrieves the position of the Message that is dequeued.</td>
</tr>
<tr>
<td>getTransformation() on page 13-70</td>
<td>Retrieves the transformation applied before a Message is dequeued.</td>
</tr>
<tr>
<td>getVisibility() on page 13-70</td>
<td>Retrieves the transactional behavior of the dequeue operation.</td>
</tr>
<tr>
<td>getWaitTime() on page 13-70</td>
<td>Retrieves the specified behavior of the Consumer when waiting for a Message with matching search criteria.</td>
</tr>
<tr>
<td>isNull() on page 13-70</td>
<td>Tests whether the Consumer object is NULL.</td>
</tr>
<tr>
<td>operator=() on page 13-70</td>
<td>Assignment operator for the Consumer class.</td>
</tr>
<tr>
<td>receive() on page 13-71</td>
<td>Receives and dequeues a Message.</td>
</tr>
<tr>
<td>setAgent() on page 13-71</td>
<td>Sets the Agent's name and address (queue name) on the consumer.</td>
</tr>
<tr>
<td>setConsumerName() on page 13-71</td>
<td>Sets the Consumer name.</td>
</tr>
<tr>
<td>setCorrelationId() on page 13-72</td>
<td>Specifies the correlation identifier of the message to be dequeued.</td>
</tr>
<tr>
<td>setDequeueMode() on page 13-72</td>
<td>Specifies the locking behavior associated with dequeuing.</td>
</tr>
<tr>
<td>setMessageIdToDequeue() on page 13-72</td>
<td>Specifies the identifier of the Message to be dequeued.</td>
</tr>
<tr>
<td>setNull() on page 13-72</td>
<td>Nullifies the Consumer; frees the memory associated with this object.</td>
</tr>
<tr>
<td>setPositionOfMessage() on page 13-73</td>
<td>Specifies position of the Message to be retrieved.</td>
</tr>
<tr>
<td>setQueueName() on page 13-73</td>
<td>Specifies the name of a queue before dequeuing Messages.</td>
</tr>
<tr>
<td>setTransformation() on page 13-73</td>
<td>Specifies transformation applied before dequeuing a Message.</td>
</tr>
<tr>
<td>setVisibility() on page 13-73</td>
<td>Specifies if Message should be dequeued as part of the current transaction.</td>
</tr>
<tr>
<td>setWaitTime() on page 13-74</td>
<td>Specifies wait conditions if there are no Messages with matching criteria.</td>
</tr>
</tbody>
</table>

---

**Consumer()**

Consumer class constructor.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer( const Connection *conn);</td>
<td>Creates a new Consumer object with the specified Connection handle.</td>
</tr>
<tr>
<td>Consumer( const Connection *conn const Agent&amp; agent);</td>
<td>Creates a new Consumer object with specified Connection and properties of the specified Agent.</td>
</tr>
<tr>
<td>Consumer( const Connection *conn, const string&amp; queue);</td>
<td>Creates a new Consumer object with specified Connection and queue.</td>
</tr>
</tbody>
</table>
getConsumerName()

Retrieves the name of the Consumer.

**Syntax**

```cpp
string getConsumerName() const;
```

getCorrelationId()

Retrieves the correlation id of the message that is to be dequeued.

**Syntax**

```cpp
string getCorrelationId() const;
```

gerDequeueMode()

Retrieves the dequeue mode of the Consumer. DequeueMode is defined in Table 13-14 on page 13-67.

**Syntax**

```cpp
DequeueMode getDequeueMode() const;
```

gerMessageIdToDequeue()

Retrieves the id of the message that is dequeued.

**Syntax**

```cpp
Bytes getMessageIdToDequeue() const;
```

gerPositionOfMessage()

Retrieves the position, or navigation, of the message that is dequeued. Navigation is defined in Table 13-14 on page 13-67.
getQueueName()

Syntax

String getQueueName() const;

getQueueName()

Gets the name of the queue used by the consumer.

Syntax

String getQueueName() const;

getTransformation()

Retrieves the transformation applied before a Message is dequeued.

Syntax

String getTransformation() const;

getVisibility()

Retrieves the transactional behavior of the dequeue operation, or visibility. Visibility is defined in Table 13–14 on page 13-67.

Syntax

Visibility getVisibility() const;

getWaitTime()

Retrieves the specified behavior of the Consumer when waiting for a Message with matching search criteria. DequeWaitOption is defined in Table 13–14 on page 13-67.

Syntax

DequeWaitOption getWaitTime() const;

isNull()

Tests whether the Consumer object is NULL. If the Consumer object is NULL, TRUE is returned; otherwise, FALSE is returned.

Syntax

Bool isNull() const;

operator=()

Assignment operator for Consumer class.

Syntax

Void operator=(
    const Consumer& consumer);
receive()

Receives and dequeues a Message.

**Syntax**

```c++
Message receive(
    Message::PayloadType pType,
    const string& type="",
    const string& schema="");
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>consumer</td>
<td>The original Consumer.</td>
</tr>
</tbody>
</table>

setAgent()

Sets the Agent’s name and address (queue name) on the consumer.

**Syntax**

```c++
void setAgent(
    const Agent& agent);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent</td>
<td>Name of the Agent.</td>
</tr>
</tbody>
</table>

setConsumerName()

Sets the Consumer name. Only messages with matching consumer name can be accessed. If a queue is not set up for multiple consumer, this option should be set to NULL.

**Syntax**

```c++
void setConsumerName(
    const string& name);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the Consumer.</td>
</tr>
</tbody>
</table>
setCorrelationId()

Specifies the correlation identifier of the message to be dequeued. Special pattern matching characters, such as the percent sign (%) and the underscore(_) can be used. If several messages satisfy the pattern, the order of dequeuing is undetermined.

**Syntax**

```cpp
void setCorrelationId
    const string& id);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The identifier of the Message.</td>
</tr>
</tbody>
</table>

setDequeueMode()

Specifies the locking behavior associated with dequeuing.

**Syntax**

```cpp
void setDequeueMode(
    DequeueMode mode);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>Behavior of enqueuing. DequeMode is defined in Table 13–14 on page 13-67.</td>
</tr>
</tbody>
</table>

setMessageIdToDequeue()

Specifies the identifier of the Message to be dequeued.

**Syntax**

```cpp
void setMessageIdToDequeue(
    const Bytes& msgid);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgid</td>
<td>Identifier of the Message to be dequeued.</td>
</tr>
</tbody>
</table>

setNull()

Nullifies the Consumer; frees the memory associated with this object.

**Syntax**

```cpp
void setNull();
```
setPositionOfMessage()

Specifies position of the Message to be retrieved.

**Syntax**

```cpp
void setPositionOfMessage(
    Navigation pos);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>Position of the message, Navigation, is defined in Table 13-14 on page 13-67.</td>
</tr>
</tbody>
</table>

setQueueName()

Specifies the name of a queue before dequeuing Messages. Typically used when dequeuing multiple messages from the same queue.

**Syntax**

```cpp
void setQueueName(
    const string& queue);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>The name of a valid queue in the database.</td>
</tr>
</tbody>
</table>

setTransformation()

Specifies transformation applied before dequeuing the Message.

**Syntax**

```cpp
void setTransformation(
    string &fName);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fName</td>
<td>SQL transformation function.</td>
</tr>
</tbody>
</table>

setVisibility()

Specifies if Message should be dequeued as part of the current transaction. Visibility parameter is ignored when in DEQ_BROWSE mode.

**Syntax**

```cpp
void setVisibility(
    Visibility option);
```
**setWaitTime()**

Specifies wait conditions if there are no *messages* with matching criteria. The `wait` parameter is ignored if messages in the same group are being dequeued.

**Syntax**

```c
void setWaitTime(
    DequeWaitOption wait);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>option</td>
<td>Visibility option being set, defined in Table 13–14 on page 13-67.</td>
</tr>
<tr>
<td>wait</td>
<td>Waiting conditions. DequeWaitOption is defined in Table 13–14 on page 13-67.</td>
</tr>
</tbody>
</table>
The Date class specifies the abstraction for a SQL DATE data item. The Date class also adds formatting and parsing operations to support the OCCI escape syntax for date values.

Since the SQL standard DATE is a subset of Oracle Date, this class can be used to support both.

Objects from the Date class can be used as standalone class objects in client side numeric computations and also used to fetch from, and set to, the database.

**Example 13–5  How to Get a Date from Database and Use it in Standalone Calculations**

This example demonstrates a Date column value being retrieved from the database, a bind using a Date object, and a computation using a standalone Date object.

```c
/* Create a connection */
Environment *env = Environment::createEnvironment(Environment::DEFAULT);
Connection *conn = Connection(user, passwd, db);

/* Create a statement and associate a DML statement to it */
string sqlStmt = "SELECT job-id, start_date from JOB_HISTORY
    where end_date = :x";
Statement *stmt = conn->createStatement(sqlStmt);

/* Create a Date object and bind it to the statement */
Date edate(env, 2000, 9, 3, 23, 30, 30);
stmt->setDate(1, edate);
ResultSet *rset = stmt->executeQuery();

/* Fetch a date from the database */
while(rset->next())
{
    Date sd = rset->getDate(2);
    Date temp = sd;  /*assignment operator */
    /* Methods on Date */
    temp.getDate(year, month, day, hour, minute, second);
    temp.setMonths(2);
    IntervalDS inter = temp.daysBetween(sd);
    
}
```

**Table 13–16  Summary of Date Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date() on page 13-76</td>
<td>Date class constructor.</td>
</tr>
<tr>
<td>addDays() on page 13-77</td>
<td>Returns a Date object with ( n ) days added.</td>
</tr>
<tr>
<td>addMonths() on page 13-77</td>
<td>Returns a Date object with ( n ) months added.</td>
</tr>
<tr>
<td>daysBetween() on page 13-77</td>
<td>Returns the number of days between the current Date object and the date specified.</td>
</tr>
<tr>
<td>fromBytes() on page 13-77</td>
<td>Convert an external Bytes representation of a Date object to a Date object.</td>
</tr>
</tbody>
</table>
### Date() Syntax Constructor

**Summary**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date();</td>
<td>Creates a NULL Date object.</td>
</tr>
<tr>
<td>Date(const Date &amp;srcDate);</td>
<td>Creates a copy of a Date object.</td>
</tr>
<tr>
<td>Date(const Environment *envp, int year = 1, unsigned int month = 1, unsigned int day = 1, unsigned int hour = 0, unsigned int minute = 0, unsigned int seconds = 0);</td>
<td>Creates a Date object using integer parameters.</td>
</tr>
</tbody>
</table>
addDays()

Adds a specified number of days to the Date object and returns the new date.

Syntax
Date addDays(
    int val) const;

Parameter Description
val The number of days to be added to the current Date object.

addMonths()

Adds a specified number of months to the Date object and returns the new date.

Syntax
Date addMonths(
    int val) const;

Parameter Description
val The number of months to be added to the current Date object.

daysBetween()

Returns the number of days between the current Date object and the date specified.

Syntax
IntervalDS daysBetween(
    const Date &date) const;

Parameter Description
date The date to be used to compute the days between.

fromBytes()

Converts a Bytes object to a Date object.
fromText()

**Syntax**

```cpp
void fromBytes(
    const Bytes &byteStream,
    const Environment *envp = NULL);
```

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byteStream</td>
<td>Date in external format in the form of Bytes.</td>
</tr>
<tr>
<td>envp</td>
<td>The OCCI environment.</td>
</tr>
</tbody>
</table>

**fromText()**

Sets Date object to value represented by a string or UString.

The value is interpreted based on the fmt and nlsParam parameters. In cases where nlsParam is not passed, the Globalization Support settings of the envp parameter are used.

**See Also:** *Oracle Database SQL Language Reference* for information on **TO_DATE**

### Syntax Description

```cpp
void fromText(
    const string &datestr,
    const string &fmt = "",
    const string &nlsParam = "",
    const Environment *envp = NULL);
```

Sets Date object to value represented by a string.

```cpp
void fromText(
    const UString &datestr,
    const UString &fmt,
    const UString &nlsParam,
    const Environment *envp = NULL);
```

Sets Date object to value represented by a UString; globalization enabled.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>envp</td>
<td>The OCCI environment.</td>
</tr>
<tr>
<td>datestr</td>
<td>The date string to be converted to a Date object.</td>
</tr>
<tr>
<td>fmt</td>
<td>The format string; default is DD-MON-YY.</td>
</tr>
<tr>
<td>nlsParam</td>
<td>The NLS parameters string.</td>
</tr>
</tbody>
</table>

**getDateTime()**

Returns the date in the form of the date components year, month, day, hour, minute, seconds.

**Syntax**

```cpp
void getDate(
    // Parameter Description
    // envp          | The OCCI environment.                            |
    // datestr      | The date string to be converted to a Date object.|
    // fmt          | The format string; default is DD-MON-YY.        |
    // nlsParam     | The NLS parameters string.                      |
```

13-78  Oracle C++ Call Interface Programmer's Guide
int &year,
unsigned int &month,
unsigned int &day,
unsigned int &hour,
unsigned int &min,
unsigned int &seconds) const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>The year component of the date.</td>
</tr>
<tr>
<td>month</td>
<td>The month component of the date.</td>
</tr>
<tr>
<td>day</td>
<td>The day component of the date.</td>
</tr>
<tr>
<td>hour</td>
<td>The hour component of the date.</td>
</tr>
<tr>
<td>min</td>
<td>The minutes component of the date.</td>
</tr>
<tr>
<td>seconds</td>
<td>The seconds component of the date.</td>
</tr>
</tbody>
</table>

**getSystemDate()**

Returns the system date.

**Syntax**

```cpp
static Date getSystemDate(
    const Environment *envp);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>envp</td>
<td>The environment in which the system date is returned.</td>
</tr>
</tbody>
</table>

**isNull()**

Tests whether the Date is NULL. If the Date is NULL, TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool isNull() const;
```

**lastDay()**

Returns a date representing the last day of the current month.

**Syntax**

```cpp
Date lastDay() const;
```

**nextDay()**

Returns a date representing the day after the day of the week specified.

**See Also:** *Oracle Database SQL Language Reference* for information on *TO_DATE*
Assigns the date object on the right side of the equal (=) sign to the date object on the left side of the equal (=) sign.

Syntax

```cpp
Date& operator=(
    const Date &d);
```

Parameter Description

dow A string representing the day of the week.

Compares the dates specified. If the dates are equal, TRUE is returned; otherwise, FALSE is returned.

Syntax

```cpp
bool operator==(const Date &first, const Date &second);
```

Parameter Description

first The first date to be compared.
second The second date to be compared.

Compares the dates specified. If the dates are not equal then TRUE is returned; otherwise, FALSE is returned.

Syntax

```cpp
bool operator!=(const Date &first, const Date &second);
```
operator>()

Compares the dates specified. If the first date is in the future relative to the second
date then TRUE is returned; otherwise, FALSE is returned. If either date is NULL then
FALSE is returned. If the dates are of different type, then FALSE is returned.

Syntax

```cpp
bool operator>(
    const Date &first,
    const Date &second);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first date to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second date to be compared.</td>
</tr>
</tbody>
</table>

operator>=()

Compares the dates specified. If the first date is in the future relative to the second
date or the dates are equal then TRUE is returned; otherwise, FALSE is returned. If either date is NULL then FALSE is returned. If the dates are of a different type, then FALSE is returned.

Syntax

```cpp
bool operator>=(
    const Date &first,
    const Date &second);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first date to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second date to be compared.</td>
</tr>
</tbody>
</table>

operator<()

Compares the dates specified. If the first date precedes the second date, then TRUE is
returned; otherwise, FALSE is returned. If either date is NULL then FALSE is returned. If the dates are of a different type, then FALSE is returned.

Syntax

```cpp
bool operator<(
    const Date &first,
    const Date &second);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first date to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second date to be compared.</td>
</tr>
</tbody>
</table>
operator<=()

Compares the dates specified. If the first date precedes the second date or the dates are equal then TRUE is returned; otherwise, FALSE is returned. If either date is NULL then FALSE is returned. If the dates are of a different type, then FALSE is returned.

**Syntax**

```cpp
bool operator<=(
    const Date &first,
    const Date &second);
```

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first date to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second date to be compared.</td>
</tr>
</tbody>
</table>

**setDate()**

Sets the date to the values specified.

**Syntax**

```cpp
void setDate(
    int year = 1,
    unsigned int month = 1,
    unsigned int day = 1,
    unsigned int hour = 0,
    unsigned int minute = 0,
    unsigned int seconds = 0);
```

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>The argument specifying the year value. Valid values are -4713 through 9999.</td>
</tr>
<tr>
<td>month</td>
<td>The argument specifying the month value. Valid values are 1 through 12.</td>
</tr>
<tr>
<td>day</td>
<td>The argument specifying the day value. Valid values are 1 through 31.</td>
</tr>
<tr>
<td>hour</td>
<td>The argument specifying the hour value. Valid values are 0 through 23.</td>
</tr>
<tr>
<td>min</td>
<td>The argument specifying the minutes value. Valid values are 0 through 59.</td>
</tr>
<tr>
<td>seconds</td>
<td>The argument specifying the seconds value. Valid values are 0 through 59.</td>
</tr>
</tbody>
</table>

**setNull()**

Sets the Date to atomically NULL.
Syntax

```c
void setNull();
```

toBytes()

Returns the date in Bytes representation.

Syntax

```c
Bytes toBytes() const;
```

toText()

Returns a string or UString with the value of this date formatted using fmt and nlsParam.

The value is interpreted based on the fmt and nlsParam parameters. In cases where nlsParam is not passed, the Globalization Support settings of the envp parameter are used.

See Also: Oracle Database SQL Language Reference for information on TO_DATE

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string toText()</td>
<td>Returns a string with the value of this date formatted using fmt and nlsParam.</td>
</tr>
<tr>
<td>const string &amp;fmt = &quot;&quot;,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;nlsParam = &quot;&quot;</td>
<td></td>
</tr>
<tr>
<td>UString toText()</td>
<td>Returns a UString with the value of this date formatted using fmt and nlsParam.</td>
</tr>
<tr>
<td>const UString &amp;fmt,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;nlsParam)</td>
<td></td>
</tr>
</tbody>
</table>

Parameter Description

- **fmt**: The format string; default is DD-MON-YY.
- **nlsParam**: The NLS parameters string. If nlsParam is specified, this determines the NLS parameters to be used for the conversion. If nlsParam is not specified, the NLS parameters are picked up from envp.

toZone()

Returns Date value converted from one time zone to another.

Syntax

```c
Date toZone(
    const string &zone1,
    const string &zone2) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zone1</td>
<td>A string representing the time zone to be converted from.</td>
</tr>
<tr>
<td>zone2</td>
<td>A string representing the time zone to be converted to.</td>
</tr>
</tbody>
</table>
Valid time zone codes are:

<table>
<thead>
<tr>
<th>Zone code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST, ADT</td>
<td>Atlantic Standard or Daylight Time</td>
</tr>
<tr>
<td>BST, BDT</td>
<td>Bering Standard or Daylight Time</td>
</tr>
<tr>
<td>CST, CDT</td>
<td>Central Standard or Daylight Time</td>
</tr>
<tr>
<td>EST, EDT</td>
<td>Eastern Standard or Daylight Time</td>
</tr>
<tr>
<td>GMT</td>
<td>Greenwich Mean Time</td>
</tr>
<tr>
<td>HST, HDT</td>
<td>Alaska-Hawaii Standard Time or Daylight Time</td>
</tr>
<tr>
<td>MST, MDT</td>
<td>Mountain Standard or Daylight Time</td>
</tr>
<tr>
<td>NST</td>
<td>Newfoundland Standard Time</td>
</tr>
<tr>
<td>PST, PDT</td>
<td>Pacific Standard or Daylight Time</td>
</tr>
<tr>
<td>YST, YDT</td>
<td>Yukon Standard or Daylight Time</td>
</tr>
</tbody>
</table>
Environment Class

The Environment class provides an OCCI environment to manage memory and other resources for OCCI objects.

The application can have multiple OCCI environments. Each environment would have its own heap and thread-safety mutexes.

Table 13–17 Enumerated Values Used by Environment Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td></td>
</tr>
<tr>
<td>DEFAULT</td>
<td>is used for creating an Environment object; it has no thread safety or object support.</td>
</tr>
<tr>
<td>OBJECT</td>
<td>is for creating an Environment object; it uses object features.</td>
</tr>
<tr>
<td>SHARED</td>
<td>is for creating an Environment object.</td>
</tr>
<tr>
<td>NO_USERCALLBACKS</td>
<td>is for creating an Environment object; it does not support user callbacks.</td>
</tr>
<tr>
<td>THREADED_MUTEXED</td>
<td>is a thread safe mode for creating an Environment object, mutexed internally by OCCI.</td>
</tr>
<tr>
<td>THREADED_UNMUTEXED</td>
<td>is a thread safe mode for creating an Environment object; the client is responsible for mutexing.</td>
</tr>
<tr>
<td>EVENTS</td>
<td>supports registration for event notification used in Oracle Streams Advanced Queuing.</td>
</tr>
<tr>
<td>USE_LDAP</td>
<td>supports registration with LDAP.</td>
</tr>
</tbody>
</table>

Table 13–18 Summary of Environment Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>createConnection()</td>
<td>Establishes a connection to the specified database.</td>
</tr>
<tr>
<td>createConnectionPool()</td>
<td>Creates a connection pool.</td>
</tr>
<tr>
<td>createEnvironment()</td>
<td>Creates an Environment object.</td>
</tr>
<tr>
<td>createStatelessConnectionPool()</td>
<td>Creates a stateless connection pool.</td>
</tr>
<tr>
<td>enableSubscription()</td>
<td>Enables subscription notification</td>
</tr>
<tr>
<td>disableSubscription()</td>
<td>Disables subscription notification</td>
</tr>
<tr>
<td>getCacheMaxSize()</td>
<td>Retrieves the Cache Max heap size.</td>
</tr>
<tr>
<td>getCacheOptSize()</td>
<td>Retrieves the cache optimal heap size.</td>
</tr>
<tr>
<td>getCacheSortedFlush()</td>
<td>Retrieves the setting of the cache sorting flag.</td>
</tr>
<tr>
<td>getClientVersion()</td>
<td>Returns the version of the client library.</td>
</tr>
<tr>
<td>getCurrentHeapSize()</td>
<td>Returns the current amount of memory allocated to all objects in the current environment.</td>
</tr>
<tr>
<td>getLDAPAdminContext()</td>
<td>Returns the administrative context when using LDAP open notification registration.</td>
</tr>
</tbody>
</table>
**createConnection()**

This method establishes a connection to the database specified.

**Syntax**

```cpp
Connection * createConnection(
    const string &userName,
    const string &password,
    const string &connectString="\"");
```

**Description**

Creates a default connection; if the `userName` and `password` are NULL, the connection may be authenticated externally.

---

**Table 13–18  (Cont.) Summary of Environment Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLDAPAuthentication() on page 13-91</td>
<td>Returns the authentication mode when using LDAP open notification registration.</td>
</tr>
<tr>
<td>getLDAPHost() on page 13-91</td>
<td>Returns the host on which the LDAP server runs.</td>
</tr>
<tr>
<td>getLDAPPort() on page 13-91</td>
<td>Returns the port on which the LDAP server is listening.</td>
</tr>
<tr>
<td>getMap() on page 13-91</td>
<td>Returns the Map for the current environment.</td>
</tr>
<tr>
<td>getOCIEnvironment() on page 13-92</td>
<td>Returns the OCI environment associated with the current environment.</td>
</tr>
<tr>
<td>getXACredentialsConnection() on page 13-92</td>
<td>Creates an XA connection to a database.</td>
</tr>
<tr>
<td>getXAEnvironment() on page 13-92</td>
<td>Creates an XA Environment object.</td>
</tr>
<tr>
<td>releaseXAConnection() on page 13-92</td>
<td>Releases all resources allocated by a <code>getXACredentialsConnection()</code> call.</td>
</tr>
<tr>
<td>releaseXAEnvironment() on page 13-93</td>
<td>Releases all resources allocated by a <code>getXAEnvironment()</code> call.</td>
</tr>
<tr>
<td>setCacheMaxSize() on page 13-93</td>
<td>Specifies the maximum size for the client-side object cache as a percentage of the optimal size.</td>
</tr>
<tr>
<td>setCacheOptSize() on page 13-93</td>
<td>Specifies the optimal size for the client-side object cache in bytes.</td>
</tr>
<tr>
<td>setCacheSortedFlush() on page 13-93</td>
<td>Specifies whether to sort cache in table order before flushing.</td>
</tr>
<tr>
<td>setLDAPAdminContext() on page 13-94</td>
<td>Specifies the administrative context for the LDAP client.</td>
</tr>
<tr>
<td>setLDAPAuthentication() on page 13-94</td>
<td>Specifies the LDAP authentication mode.</td>
</tr>
<tr>
<td>setLDAPHostAndPort() on page 13-94</td>
<td>Specifies the LDAP server host and port.</td>
</tr>
<tr>
<td>setLDAPLoginNameAndPassword() on page 13-95</td>
<td>Specifies the login name and password when connecting to an LDAP server.</td>
</tr>
<tr>
<td>terminateConnection() on page 13-95</td>
<td>Closes the connection pool and free all related resources.</td>
</tr>
<tr>
<td>terminateConnectionPool() on page 13-95</td>
<td>Closes the connection pool and free all related resources.</td>
</tr>
<tr>
<td>terminateEnvironment() on page 13-95</td>
<td>Destroys the environment.</td>
</tr>
<tr>
<td>terminateStatelessConnectionPool() on page 13-96</td>
<td>Closes the stateless connection pool and free all related resources.</td>
</tr>
</tbody>
</table>
**createConnectionPool()**

Creates a connection pool based on the parameters specified.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionPool* createConnectionPool(</td>
<td></td>
</tr>
<tr>
<td>const string &amp;poolUserName,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;poolPassword,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;connectString = &quot;&quot;,</td>
<td></td>
</tr>
<tr>
<td>unsigned int minConn = 0,</td>
<td></td>
</tr>
<tr>
<td>unsigned int maxConn = 1,</td>
<td></td>
</tr>
<tr>
<td>unsigned int incrConn = 1)=0;</td>
<td>Creates a default connection pool.</td>
</tr>
<tr>
<td>ConnectionPool* createConnectionPool(</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;poolUserName,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;poolPassword,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;connectString,</td>
<td></td>
</tr>
<tr>
<td>unsigned int minConn = 0,</td>
<td></td>
</tr>
<tr>
<td>unsigned int maxConn = 1,</td>
<td></td>
</tr>
<tr>
<td>unsigned int incrConn = 1)=0;</td>
<td>Creates a connection pool (Unicode support). The client Environment should be initialized in OCCIUTIF16 mode.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>poolUserName</td>
<td>The pool user name.</td>
</tr>
</tbody>
</table>
createEnvironment()

Creates an Environment object. It is created with the specified memory management functions specified in the setMemMgrFunctions() method. If no memory manager functions are specified, then OCCI uses its own default functions. An Environment object must eventually be closed to free all the system resources it has acquired.

If the Mode is specified is either THREADED_MUTEXED or THREADED_UNMUTEXED as defined in Table 13–17 on page 13-85, then all three memory management functions must be thread-safe.

Parameter Description
---
poolPassword The pool password.
connectString The connection string for the server
minConn The minimum number of connections in the pool. The minimum number of connections are opened by this method. Additional connections are opened only when necessary. Generally, minConn should be set to the number of concurrent statements the application is expected to run.
maxConn The maximum number of connections in the pool. Valid values are 1 and greater.
incrConn The increment by which to increase the number of connections to be opened if the current number of connections is less than maxConn. Valid values are 1 and greater.

createEnvironment()

static Environment * createEnvironment{
    Mode mode = DEFAULT,
    void *ctxp = 0,
    void *(malocfp)(void *ctxp,
                  size_t size) = 0,
    void *(ralocfp)(void *ctxp,
                   void *memptr,
                   size_t newsize) = 0,
    void (*mfreefp)(void *ctxp,
                   void *memptr) = 0;

    static Environment * createEnvironment(
        const string &charset,
        const string &ncharset,
        Mode mode = DEFAULT,
        void *ctxp = 0,
        void *(malocfp)(void *ctxp,
                       size_t size) = 0,
        void *(ralocfp)(void *ctxp,
                       void *memptr,
                       size_t newsize) = 0,
        void (*mfreefp)(void *ctxp,
                        void *memptr) = 0);

Parameter Description
---
mode Values are defined as part of Mode in Table 13–17 on page 13-85: DEFAULT, THREADED_MUTEXED, THREADED_UNMUTEXED, OBJECT.
**createStatelessConnectionPool()**

Creates a StatelessConnectionPool object with specified pool attributes.

```cpp
StatelessConnectionPool* createStatelessConnectionPool(const string &poolUserName,
const string &poolPassword,
const string &connectString = "",
unsigned int maxConn = 1,
unsigned int minConn = 0,
unsigned int incrConn = 1,
StatelessConnectionPool::PoolType pType = StatelessConnectionPool::HETEROGENEOUS);  
```

Support for string.

```cpp
StatelessConnectionPool* createStatelessConnectionPool(const UString &poolUserName,
const UString &poolPassword,
const UString &connectString,
unsigned int maxConn = 1,
unsigned int minConn = 0,
unsigned int incrConn = 1,
StatelessConnectionPool::PoolType pType = StatelessConnectionPool::HETEROGENEOUS);  
```

Support for UString.

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>poolUserName</td>
<td>The pool user name.</td>
</tr>
<tr>
<td>poolPassword</td>
<td>The pool password.</td>
</tr>
<tr>
<td>connectString</td>
<td>The connection string for the server.</td>
</tr>
<tr>
<td>maxConn</td>
<td>The maximum number of connections that can be opened the pool; additional sessions cannot be open.</td>
</tr>
<tr>
<td>minConn</td>
<td>The number of connections initially created in a pool. This parameter is considered only if the PoolType is set to HOMOGENEOUS, as defined in Table 13–41 on page 13-203.</td>
</tr>
</tbody>
</table>
enableSubscription()

Enables subscription notification.

**Syntax**

```cpp
void enableSubscription(
    const aq::Subscription &sub);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>The Subscription.</td>
</tr>
</tbody>
</table>

disableSubscription()

Disables subscription notification.

**Syntax**

```cpp
void disableSubscription(
    Subscription &subscr);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscr</td>
<td>The Subscription.</td>
</tr>
</tbody>
</table>

getCacheMaxSize()

Retrieves the maximum size of the cache.

**Syntax**

```cpp
unsigned int getCacheMaxSize() const;
```

gCacheOptSize()

Retrieves the Cache optimal heap size.

**Syntax**

```cpp
unsigned int getCacheOptSize() const;
```
**getCacheSortedFlush()**

Retrieves the current setting of the cache sorting flag; TRUE or FALSE.

**Syntax**

```cpp
bool getCacheSortedFlush() const;
```

**getCurrentHeapSize()**

Returns the amount of memory currently allocated to all objects in this environment.

**Syntax**

```cpp
unsigned int getCurrentHeapSize() const;
```

**getLDAPAdminContext()**

Returns the administrative context when using LDAP open notification registration.

**Syntax**

```cpp
string getLDAPAdminContext() const;
```

**getLDAPAuthentication()**

Returns the authentication mode when using LDAP open notification registration.

**Syntax**

```cpp
unsigned int getLDAPAuthentication() const;
```

**getLDAPHost()**

Returns the host on which the LDAP server runs.

**Syntax**

```cpp
string getLDAPHost() const;
```

**getLDAPPort()**

Returns the port on which the LDAP server is listening.

**Syntax**

```cpp
unsigned int getLDAPPort() const;
```

**getMap()**

Returns a pointer to the map for this environment.
getOCIEnvironment()

Returns a pointer to the OCI environment associated with this environment.

Syntax

Map *getMap() const;

getOCIEnvironment()

Returns a pointer to the OCI environment associated with this environment.

Syntax

OCIEnv *getOCIEnvironment() const;

getXAConnection()

Returns a pointer to an OCCI Connection object that corresponds to the one opened by the XA library.

Syntax

Connection* getXAConnection(const string &dbname);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbname</td>
<td>The database name; same as the optional dbname provided in the Open String (and used in connection to the Resource Manager).</td>
</tr>
</tbody>
</table>

getXAEnvironment()

Returns a pointer to an OCCI Environment object that corresponds to the one opened by the XA library.

Syntax

Environment *getXAEnvironment(const string &dbname);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbname</td>
<td>The database name; same as the optional dbname provided in the Open String (and used in connection to the Resource Manager).</td>
</tr>
</tbody>
</table>

releaseXAConnection()

Release/deallocate all resources allocated by the getXAConnection() method.

Syntax

void releaseXAConnection(Connection* conn);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>The connection returned by the getXAConnection() method.</td>
</tr>
</tbody>
</table>
releaseXAEnvironment()

Release/deallocate all resources allocated by the getXAEnvironment() method.

**Syntax**

```c
void releaseXAEnvironment(
    Environment* env);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>The environment returned by the getXAEnvironment() method.</td>
</tr>
</tbody>
</table>

setCacheMaxSize()

Sets the maximum size for the client-side object cache as a percentage of the optimal size. The default value is 10%.

**Syntax**

```c
void setCacheMaxSize(
    unsigned int maxSize);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxSize</td>
<td>The value of the maximum size, as a percentage.</td>
</tr>
</tbody>
</table>

setCacheOptSize()

Sets the optimal size for the client-side object cache in bytes. The default value is 8MB.

**Syntax**

```c
void setCacheOptSize(
    unsigned int optSize);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optSize</td>
<td>The value of the optimal size, in bytes.</td>
</tr>
</tbody>
</table>

setCacheSortedFlush()

Sets the cache flushing protocol. By default, objects in cache are flushed in the order they are modified; **flag=FALSE**. To improve server-side performance, set **flag=TRUE**, so that the objects in cache are sorted in table order before flushing from client cache.

**Syntax**

```c
void setCacheSortedFlush(
    bool flag);
```
setLDAPAdminContext()

Sets the administrative context of the client. This is usually the root of the Oracle RDBMS LDAP schema in the LDAP server.

**Syntax**

```cpp
void setLDAPAdminContext(
    const string &ctx);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>The client context</td>
</tr>
</tbody>
</table>

**setLDAPAuthentication()**

Specifies the authentication mode. Currently the only supported value is 0x1: Simple authentication; username/password authentication.

**Syntax**

```cpp
void setLDAPAuthentication(
    unsigned int mode);`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>The authentication mode</td>
</tr>
</tbody>
</table>

**setLDAPHostAndPort()**

Specifies the host on which the LDAP server is running, and the port on which it is listening for requests.

**Syntax**

```cpp
void setLDAPHostAndPort(
    const string &host,
    unsigned int port);`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>The host for LDAP</td>
</tr>
<tr>
<td>port</td>
<td>The port for LDAP</td>
</tr>
</tbody>
</table>
setLDAPLoginNameAndPassword()

Specifies the login distinguished name and password used when connecting to an LDAP server.

**Syntax**

```cpp
void setLDAPLoginNameAndPassword(
    const string &login,
    const &passwd);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>login</td>
<td>The login name</td>
</tr>
<tr>
<td>passwd</td>
<td>The login password</td>
</tr>
</tbody>
</table>

terminateConnection()

Closes the connection to the environment, and frees all related system resources.

**Syntax**

```cpp
void terminateConnection(
    Connection *connection);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>A pointer to the connection instance to be terminated.</td>
</tr>
</tbody>
</table>

terminateConnectionPool()

Closes the connections in the connection pool, and frees all related system resources.

**Syntax**

```cpp
void terminateConnectionPool(
    ConnectionPool *poolPointer);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>poolPointer</td>
<td>A pointer to the connection pool instance to be terminated.</td>
</tr>
</tbody>
</table>

terminateEnvironment()

Closes the environment, and frees all related system resources.

**Syntax**

```cpp
void terminateEnvironment(
    Environment *env);
```
terminateStatelessConnectionPool()

Destroys the specified StatelessConnectionPool.

**Syntax**

```c
void terminateStatelessConnectionPool(
    StatelessConnectionPool* poolPointer,
    StatelessConnectionPool::DestroyMode mode=StatelessConnectionPool::DEFAULT);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>Environment to be closed.</td>
</tr>
<tr>
<td>poolPointer</td>
<td>The StatelessConnectionPool to be destroyed.</td>
</tr>
<tr>
<td>mode</td>
<td>DestroyMode as defined Table 13–41 on page 13-203: DEFAULT or SPF_FORCE.</td>
</tr>
</tbody>
</table>
IntervalDS Class

The IntervalDS class encapsulates time interval calculations in terms of days, hours, minutes, seconds, and fractional seconds. Leading field precision is determined by number of decimal digits in day input. Fraction second precision is determined by number of fraction digits on input.

Table 13–19 Fields of IntervalDS Class

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>day</td>
<td>int</td>
<td>Day component. Valid values are (-10^9) through (10^9).</td>
</tr>
<tr>
<td>hour</td>
<td>int</td>
<td>Hour component. Valid values are (-23) through (23).</td>
</tr>
<tr>
<td>minute</td>
<td>int</td>
<td>Minute component. Valid values are (-59) through (59).</td>
</tr>
<tr>
<td>second</td>
<td>int</td>
<td>Second component. Valid values are (-59) through (59).</td>
</tr>
<tr>
<td>fs</td>
<td>int</td>
<td>Fractional second component. Constructs a NULL IntervalDS object. A NULL intervalDS can be initialized by assignment or calling fromText method. Methods that can be called on NULL intervalDS objects are setName() and isNull().</td>
</tr>
</tbody>
</table>

Example 13–6 How to Use an Empty IntervalDS Object through Direct Assignment

This example demonstrates how the default constructor creates a NULL value, and how you can assign a non NULL value to a day-second interval and then perform operations on it.

```cpp
Environment *env = Environment::createEnvironment();

// Create a NULL day-second interval
IntervalDS ds;
if(ds.isNull())
  cout << "ds is null";

// Assign a non-NULL value to ds
IntervalDS anotherDS(env, "10 20:14:10.2");
ds = anotherDS;

// Now all operations on IntervalDS are valid
int DAY = ds.getDay();
```

Example 13–7 How to Use an Empty IntervalDS Object Through *Text() Methods

This example demonstrates how to create a NULL day-second interval, initialize the day-second interval by using the fromText() method, add to the day-second interval by using the += operator, multiply by using the * operator, compare 2 day-second intervals, and convert a day-second interval to a string by using the toText method:

```cpp
Environment *env = Environment::createEnvironment();

// Create a null day-second interval
IntervalDS ds1

// Initialize a null day-second interval by using the fromText method
ds1.fromText("20 10:20:30.9", env);

IntervalDS addWith(env, 2, 1);
```
ds1 += addWith;  //call += operator

IntervalDS mulDs1 = ds1 * Number(env,10);  
//call * operator
if(ds1==mulDs1)  //call == operator
{
    
    string strds = ds1.toText(2,4);  
    //2 is leading field precision  
    //4 is the fractional field precision

Table 13–20 Summary of IntervalDS Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntervalDS() on page 13-99</td>
<td>IntervalDS class constructor.</td>
</tr>
<tr>
<td>fromText() on page 13-99</td>
<td>Returns an IntervalDS converted from a string.</td>
</tr>
<tr>
<td>fromUText() on page 13-100</td>
<td>Returns an IntervalDS converted from a UString.</td>
</tr>
<tr>
<td>getDay() on page 13-100</td>
<td>Returns day interval values.</td>
</tr>
<tr>
<td>getFracSec() on page 13-100</td>
<td>Returns fractional second interval values.</td>
</tr>
<tr>
<td>getFracSec() on page 13-100</td>
<td>Returns hour interval values.</td>
</tr>
<tr>
<td>getMinute() on page 13-100</td>
<td>Returns minute interval values.</td>
</tr>
<tr>
<td>getSecond() on page 13-101</td>
<td>Returns second interval values.</td>
</tr>
<tr>
<td>isnull() on page 13-101</td>
<td>Returns true if IntervalDS is NULL, false otherwise.</td>
</tr>
<tr>
<td>operator*() on page 13-101</td>
<td>Returns the product of two IntervalDS values.</td>
</tr>
<tr>
<td>operator*=() on page 13-101</td>
<td>Multiplication assignment.</td>
</tr>
<tr>
<td>operator=() on page 13-101</td>
<td>Simple assignment.</td>
</tr>
<tr>
<td>operator==() on page 13-102</td>
<td>Checks if a and b are equal.</td>
</tr>
<tr>
<td>operator!=() on page 13-102</td>
<td>Checks if a and b are not equal.</td>
</tr>
<tr>
<td>operator/() on page 13-102</td>
<td>Returns an IntervalDS with value (a / b).</td>
</tr>
<tr>
<td>operator/=() on page 13-103</td>
<td>Division assignment.</td>
</tr>
<tr>
<td>operator&lt;() on page 13-103</td>
<td>Checks if a is greater than b.</td>
</tr>
<tr>
<td>operator&gt;=() on page 13-103</td>
<td>Checks if a is greater than or equal to b.</td>
</tr>
<tr>
<td>operator&gt;() on page 13-103</td>
<td>Checks if a is less than b.</td>
</tr>
<tr>
<td>operator=&lt;() on page 13-104</td>
<td>Checks if a is less than or equal to b.</td>
</tr>
<tr>
<td>operator-() on page 13-104</td>
<td>Returns an IntervalDS with value (a - b).</td>
</tr>
<tr>
<td>operator-=() on page 13-104</td>
<td>Subtraction assignment.</td>
</tr>
<tr>
<td>operator+() on page 13-105</td>
<td>Returns the sum of two IntervalDS values.</td>
</tr>
<tr>
<td>operator+=() on page 13-105</td>
<td>Addition assignment.</td>
</tr>
<tr>
<td>set() on page 13-105</td>
<td>Sets day-second interval.</td>
</tr>
<tr>
<td>setNull() on page 13-106</td>
<td>Sets day-second interval to NULL.</td>
</tr>
<tr>
<td>toText() on page 13-106</td>
<td>Converts to a string representation for the interval.</td>
</tr>
<tr>
<td>toUText() on page 13-106</td>
<td>Converts to a UString representation for the interval.</td>
</tr>
</tbody>
</table>
IntervalDS()

IntervalDS class constructor.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntervalDS();</td>
<td>Constructs a NULL IntervalDS object. A NULL IntervalDS can be initialized by assignment or calling fromText() method. Methods that can be called on NULL IntervalDS objects are setName() and isNull().</td>
</tr>
<tr>
<td>IntervalDS( const Environment *env, int day = 0, int hour = 0, int minute = 0, int second = 0, int fs = 0);</td>
<td>Constructs an IntervalDS object within a specified Environment.</td>
</tr>
<tr>
<td>IntervalDS( const IntervalDS &amp;src);</td>
<td>Constructs an IntervalYM object from src.</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>The Environment.</td>
</tr>
<tr>
<td>day</td>
<td>The day field of IntervalDS.</td>
</tr>
<tr>
<td>hour</td>
<td>The hour field of IntervalDS.</td>
</tr>
<tr>
<td>minute</td>
<td>The minute field of IntervalDS.</td>
</tr>
<tr>
<td>second</td>
<td>The second field of IntervalDS.</td>
</tr>
<tr>
<td>fs</td>
<td>The fs field of IntervalDS.</td>
</tr>
<tr>
<td>src</td>
<td>The source that the IntervalDS object is copied from.</td>
</tr>
</tbody>
</table>

fromText()

Creates the interval from the string specified. The string is converted using the nls parameters associated with the relevant environment. The nls parameters are picked up from env. If env is NULL, the nls parameters are picked up from the environment associated with the instance, if any.

Syntax

```cpp
void fromText(
    const string &inpstr,  
    const string &nlsParam = "",  
    const Environment *env = NULL);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inpstr</td>
<td>Input string representing a day second interval of the form 'days hours:minutes:seconds', for example, '10 20:14:10.2'</td>
</tr>
<tr>
<td>nlsParam</td>
<td>The NLS parameter string. If nlsParam is specified, this determines the NLS parameters to be used for the conversion. If nlsParam is not specified, the NLS parameters are picked up from envp.</td>
</tr>
</tbody>
</table>
fromUText()

Creates the interval from the **UString** specified.

**Syntax**

```c
void fromUText(
    const UString &inpstr,
    const Environment *env=NULL);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inpstr</td>
<td>Input <strong>UString</strong> representing a day second interval of the form 'days hours:minutes:seconds', for example, '10 20:14:10.2'</td>
</tr>
<tr>
<td>env</td>
<td>The <strong>Environment</strong>.</td>
</tr>
</tbody>
</table>

getDay()

Returns the day component of the interval.

**Syntax**

```c
int getDay() const;
```

getFracSec()

Returns the fractional second component of the interval.

**Syntax**

```c
int getFracSec() const;
```

gETCHour()

Returns the hour component of the interval.

**Syntax**

```c
int getHour() const;
```

getMinute()

Returns the minute component of this interval.

**Syntax**

```c
int getMinute() const;
```
getSecond()  

Returns the seconds component of this interval.

**Syntax**

```cpp
int getSecond() const;
```

isNull()  

Tests whether the interval is NULL. If the interval is NULL then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool isNull() const;
```

operator*()  

Multiplies an interval by a specified value and returns the result.

**Syntax**

```cpp
const IntervalDS operator* (  
    const IntervalDS &interval,  
    const Number &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval</td>
<td>Interval to be multiplied.</td>
</tr>
<tr>
<td>val</td>
<td>Value by which interval is to be multiplied.</td>
</tr>
</tbody>
</table>

operator*()=()  

Assigns the product of IntervalDS and a to IntervalDS.

**Syntax**

```cpp
IntervalDS& operator*= (  
    const IntervalDS &factor);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>factor</td>
<td>A day second interval.</td>
</tr>
</tbody>
</table>

operator()=()  

Assigns the specified value to the interval.

**Syntax**

```cpp
IntervalDS& operator= (  
    const IntervalDS &src);
```
operator==()

Compares the intervals specified. If the intervals are equal, then \texttt{TRUE} is returned; otherwise, \texttt{FALSE} is returned. If either interval is \texttt{NULL} then \texttt{SQLException} is thrown.

**Syntax**

```cpp
bool operator==(
    const IntervalDS &first,
    const IntervalDS &second);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>Value to be assigned.</td>
</tr>
</tbody>
</table>

operator!=( )

Compares the intervals specified. If the intervals are not equal then \texttt{TRUE} is returned; otherwise, \texttt{FALSE} is returned. If either interval is \texttt{NULL} then \texttt{SQLException} is thrown.

**Syntax**

```cpp
bool operator!=(
    const IntervalDS &first,
    const IntervalDS &second);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

operator/()

Returns the result of dividing an interval by a constant value.

**Syntax**

```cpp
const IntervalDS operator/(
    const IntervalDS &dividend,
    const Number &factor);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dividend</td>
<td>The interval to be divided.</td>
</tr>
<tr>
<td>factor</td>
<td>Value by which interval is to be divided.</td>
</tr>
</tbody>
</table>
operator/=()

Assigns the quotient of IntervalDS and val to IntervalDS.

**Syntax**

```cpp
IntervalDS& operator/=(
    const IntervalDS &factor);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>factor</td>
<td>A day second interval.</td>
</tr>
</tbody>
</table>

operator>()

Compares the intervals specified. If the first interval is greater than the second interval then TRUE is returned; otherwise, FALSE is returned. If either interval is NULL then SQLException is thrown.

**Syntax**

```cpp
bool operator>(
    const IntervalDS &first,
    const IntervalDS &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

operator>=(())

Compares the intervals specified. If the first interval is greater than or equal to the second interval then TRUE is returned; otherwise, FALSE is returned. If either interval is NULL then SQLException is thrown.

**Syntax**

```cpp
bool operator>=(
    const IntervalDS &first,
    const IntervalDS &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

operator<()(

Compares the intervals specified. If the first interval is less than the second interval then TRUE is returned; otherwise, FALSE is returned. If either interval is NULL then SQLException is thrown.
Syntax

```cpp
bool operator<(
    const IntervalDS &first,
    const IntervalDS &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

**operator<=()**

Compares the intervals specified. If the first interval is less than or equal to the second interval then `true` is returned; otherwise, `false` is returned. If either interval is `NULL` then SQLException is thrown.

Syntax

```cpp
bool operator<=(
    const IntervalDS &first,
    const IntervalDS &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

**operator-()**

Returns the difference between the intervals `first` and `second`.

Syntax

```cpp
const IntervalDS operator-(
    const IntervalDS &first,
    const IntervalDS &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

**operator-=(())**

Assigns the difference between `IntervalDS` and `val` to `IntervalDS`.

Syntax

```cpp
IntervalDS& operator-=(
    const IntervalDS &val);
```
**IntervalDS Class**

**operator+()**

Returns the sum of the intervals specified.

**Syntax**

```cpp
const IntervalDS operator+(  
    const IntervalDS &first,  
    const IntervalDS &second);  
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>A day second interval.</td>
</tr>
</tbody>
</table>

**Parameter Description**

- **first**: The first interval to be compared.
- **second**: The second interval to be compared.

**operator+=()**

Assigns the sum of IntervalDS and val to IntervalDS.

**Syntax**

```cpp
IntervalDS& operator+=(  
    const IntervalDS &val);  
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>A day second interval.</td>
</tr>
</tbody>
</table>

**set()**

Sets the interval to the values specified.

**Syntax**

```cpp
void set(  
    int day,  
    int hour,  
    int minute,  
    int second,  
    int fracsec);  
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>day</td>
<td>Day component.</td>
</tr>
<tr>
<td>hour</td>
<td>Hour component.</td>
</tr>
<tr>
<td>min</td>
<td>Minute component.</td>
</tr>
<tr>
<td>second</td>
<td>Second component.</td>
</tr>
<tr>
<td>fracsec</td>
<td>Fractional second component.</td>
</tr>
</tbody>
</table>
setNull()

Sets the IntervalDS to NULL.

Syntax

void setNull();

toText()

Converts to a string representation for the interval.

Syntax

string toText(
    unsigned int lfprec,
    unsigned int fsprec,
    const string &nlsParam = "") const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfprec</td>
<td>Leading field precision.</td>
</tr>
<tr>
<td>fsprec</td>
<td>Fractional second precision.</td>
</tr>
<tr>
<td>nlsParam</td>
<td>The NLS parameters string. If nlsParam is specified, this determines the NLS parameters to be used for the conversion. If nlsParam is not specified, the NLS parameters are picked up from envp.</td>
</tr>
</tbody>
</table>

toUText()

Converts to a UString representation for the interval.

Syntax

UString toUText(
    unsigned int lfprec,
    unsigned int fsprec) const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfprec</td>
<td>Leading field precision.</td>
</tr>
<tr>
<td>fsprec</td>
<td>Fractional second precision.</td>
</tr>
</tbody>
</table>
**IntervalYM Class**

`IntervalYM` supports the SQL standard data type Year-Month Interval. Leading field precision is determined by number of decimal digits on input.

**Example 13–8  How to Use an Empty `IntervalYM` Object Through Direct Assignment**

This example demonstrates that the default constructor creates a NULL value, and how you can assign a non NULL value to a year-month interval and then perform operations on it:

```cpp
Environment *env = Environment::createEnvironment();

// Create a NULL year-month interval
IntervalYM ym
if(ym.isNull())
    cout << "
 ym is null";

// Assign a non-NULL value to ym
IntervalYM anotherYM(env, "10-30");
ym=anotherYM;

// Now all operations on YM are valid
int yr = ym.getYear();
```

**Example 13–9  How to Use an `IntervalYM` Object Through ResultSet and `toText()` Method**

This example demonstrates how to get the year-month interval column from a result set, add to the year-month interval by using the `+=` operator, multiply by using the `*` operator, compare 2 year-month intervals, and convert a year-month interval to a string by using the `toText()` method.

```cpp
//SELECT WARRANT_PERIOD from PRODUCT_INFORMATION
//obtain result set
resultset->next();

//get interval value from resultset
IntervalYM ym1 = resultset->getIntervalYM(1);

IntervalYM addWith(env, 10, 1);
ym1 += addWith;    //call += operator

IntervalYM mulYm1 = ym1 * Number(env, 10);    //call * operator
if(ym1<mulYm1)    //comparison
    .

string strym = ym1.toText(3);    //3 is the leading field precision
```

**Table 13–21  Fields of `IntervalYM` Class**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>int</td>
<td>Year component. Valid values are $-10^9$ through $10^9$.</td>
</tr>
<tr>
<td>month</td>
<td>int</td>
<td>Month component. Valid values are $-11$ through $11$.</td>
</tr>
</tbody>
</table>
IntervalYM()

IntervalYM class constructor.

Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntervalYM();</td>
<td>Constructs a NULL IntervalYM object. A NULL IntervalYM can be initialized by assignment or calling operator*() method. Methods that can be called on NULL IntervalYM objects are setName() and isNull().</td>
</tr>
<tr>
<td>IntervalYM(const Environment *env, int year = 0, int month = 0);</td>
<td>Creates an IntervalYM object within the specified Environment.</td>
</tr>
</tbody>
</table>
IntervalYM Class

fromText()

This method initializes the interval to the values in inpstr. The string is interpreted using the NLS parameters set in the environment. The NLS parameters are picked up from env. If env is NULL, the NLS parameters are picked up from the environment associated with the instance, if any.

Syntax

```cpp
void fromText(
    const string &inpStr,
    const string &nlsParam = "",
    const Environment *env = NULL);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>The Environment.</td>
</tr>
<tr>
<td>year</td>
<td>The year field of the IntervalYM object.</td>
</tr>
<tr>
<td>month</td>
<td>The month field of the IntervalYM object.</td>
</tr>
<tr>
<td>src</td>
<td>The source that the IntervalYM object is copied from.</td>
</tr>
</tbody>
</table>

fromUText()

Creates the interval from the UString specified.

Syntax

```cpp
void fromUText(
    const UString &inpStr,
    const Environment *env = NULL);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inpStr</td>
<td>Input UString representing a year month interval of the form 'year-month'.</td>
</tr>
<tr>
<td>nlsParam</td>
<td>The NLS parameters string. If nlsParam is specified, this determines the NLS parameters to be used for the conversion. If nlsParam is not specified, the NLS parameters are picked up from envp.</td>
</tr>
<tr>
<td>env</td>
<td>Environment whose NLS parameters are used.</td>
</tr>
</tbody>
</table>

Parameter Description

env The Environment.
year The year field of the IntervalYM object.
month The month field of the IntervalYM object.
src The source that the IntervalYM object is copied from.
getMonth()

This method returns the month component of the interval.

**Syntax**

```cpp
int getMonth() const;
```

getYear()

This method returns the year component of the interval.

**Syntax**

```cpp
int getYear() const;
```

isNull()

This method tests whether the interval is NULL. If the interval is NULL then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool isNull() const;
```

operator*()

This method multiplies the interval by a factor and returns the result.

**Syntax**

```cpp
const IntervalYM operator*(
    const IntervalDS &interval
    const Number &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval</td>
<td>Interval to be multiplied.</td>
</tr>
<tr>
<td>val</td>
<td>Value by which interval is to be multiplied.</td>
</tr>
</tbody>
</table>

operator*()=()

This method multiplies the interval by a specified value.

**Syntax**

```cpp
IntervalYM& operator*(
    const Number &factor);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>factor</td>
<td>Value to be multiplied.</td>
</tr>
</tbody>
</table>
**operator=()**

This method assigns the specified value to the interval.

**Syntax**

```cpp
IntervalYM& operator=(
    const IntervalYM &src);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>Value to be assigned.</td>
</tr>
</tbody>
</table>

**operator==()**

This method compares the intervals specified. If the intervals are equal then `TRUE` is returned; otherwise, `FALSE` is returned. If either interval is `NULL` then `SQLException` is thrown.

**Syntax**

```cpp
bool operator==(  
    const IntervalYM &first,  
    const IntervalYM &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

**operator!=()**

This method compares the intervals specified. If the intervals are not equal then `TRUE` is returned; otherwise, `FALSE` is returned. If either interval is `NULL` then `SQLException` is thrown.

**Syntax**

```cpp
bool operator!=(  
    const IntervalYM &first,  
    const IntervalYM &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

**operator/()**

This method returns the result of dividing the interval by a factor.
operator/()  

Syntax

```cpp
const IntervalYM operator/(
    const IntervalYM &dividend,
    const Number &factor);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dividend</td>
<td>The interval to be divided.</td>
</tr>
<tr>
<td>factor</td>
<td>Value by which interval is to be divided.</td>
</tr>
</tbody>
</table>

operator/=()  

This method divides the interval by a factor.

Syntax

```cpp
IntervalYM& operator/=(
    const Number &factor);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>factor</td>
<td>A day second interval.</td>
</tr>
</tbody>
</table>

operator>()  

This method compares the intervals specified. If the first interval is greater than the second interval then TRUE is returned; otherwise, FALSE is returned. If either interval is NULL then SQLException is thrown.

Syntax

```cpp
bool operator>(
    const IntervalYM &first,
    const IntervalYM &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

operator>=(())  

This method compares the intervals specified. If the first interval is greater than or equal to the second interval then TRUE is returned; otherwise, FALSE is returned. If either interval is NULL then SQLException is thrown.

Syntax

```cpp
bool operator>=(
    const IntervalYM &first,
    const IntervalYM &second);
```
operator<()

This method compares the intervals specified. If the first interval is less than the second interval then TRUE is returned; otherwise, FALSE is returned. If either interval is NULL then SQLException is thrown.

**Syntax**

```c++
bool operator<(
    const IntervalYM &first,
    const IntervalYM &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

operator<=()

This method compares the intervals specified. If the first interval is less than or equal to the second interval then TRUE is returned; otherwise, FALSE is returned. If either interval is NULL then SQLException is thrown.

**Syntax**

```c++
bool operator<=(
    const IntervalYM &first,
    const IntervalYM &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

operator-()

This method returns the difference between the intervals specified.

**Syntax**

```c++
const IntervalYM operator-(
    const IntervalYM &first,
    const IntervalYM &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
</tbody>
</table>
operator−=()

This method computes the difference between itself and another interval.

Syntax

```cpp
IntervalYM& operator−=(
    const IntervalYM &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>A day second interval.</td>
</tr>
</tbody>
</table>

operator+()

This method returns the sum of the intervals specified.

Syntax

```cpp
const IntervalYM operator+( 
    const IntervalYM &first,
    const IntervalYM &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The first interval to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>The second interval to be compared.</td>
</tr>
</tbody>
</table>

operator+=()

This method assigns the sum of `IntervalYM` and `val` to `IntervalYM`.

Syntax

```cpp
IntervalYM& operator+=(
    const IntervalYM &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>A day second interval.</td>
</tr>
</tbody>
</table>

set()

This method sets the interval to the values specified.

Syntax

```cpp
void set(
    int year,
```
int month);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>Year component. Valid values are (-10^9) through (10^9).</td>
</tr>
<tr>
<td>month</td>
<td>Month component. Valid values are (-11) through (11).</td>
</tr>
</tbody>
</table>

**setNull()**

This method sets the interval to NULL.

**Syntax**

```cpp
void setNull();
```

**toText()**

This method returns the string representation of the interval.

**Syntax**

```cpp
string toText(
    unsigned int lfprec,
    const string &nlsParam = "") const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfprec</td>
<td>Leading field precision.</td>
</tr>
<tr>
<td>nlsParam</td>
<td>The NLS parameters string. If nlsParam is specified, this determines the NLS parameters to be used for the conversion. If nlsParam is not specified, the NLS parameters are picked up from envp.</td>
</tr>
</tbody>
</table>

**toUText()**

Converts to a UString representation for the interval.

**Syntax**

```cpp
UString toUText(
    unsigned int lfprec) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfprec</td>
<td>Leading field precision.</td>
</tr>
</tbody>
</table>
Listener Class

The Listener class encapsulates the ability to listen for Messages, on behalf of registered Agents, at specified queues.

*Table 13–23* Summary of Listener Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listener() on page 13-116</td>
<td>Listener class constructor.</td>
</tr>
<tr>
<td>getAgentList() on page 13-116</td>
<td>Retrieves the list of Agents for which the Listener provides its services.</td>
</tr>
<tr>
<td>getTimeOutForListen() on page 13-117</td>
<td>Retrieves the time out for a call.</td>
</tr>
<tr>
<td>listen() on page 13-117</td>
<td>Listens for Messages and returns the name of the Agent for whom a Message is intended.</td>
</tr>
<tr>
<td>setAgentList() on page 13-117</td>
<td>Specifies the list of Agents for which the Listener provides its services.</td>
</tr>
<tr>
<td>setTimeOutForListen() on page 13-117</td>
<td>Specifies the time out for a listen() call.</td>
</tr>
</tbody>
</table>

Listener()

Listener class constructor.

**Syntax**

```cpp
Listener(const Connection* conn);
```

Creates a Listener object.

```cpp
Listener(const Connection* conn
 vector<Agent> &aglist,
 int waitTime=0);
```

Creates a Listener object and sets the list of Agents on behalf of which it listens on queues. Also sets the waiting time; default: no waiting.

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>The connection of the new Listener object.</td>
</tr>
<tr>
<td>aglist</td>
<td>The list of agents on behalf of which the Listener object waits on queues; clients of this Listener.</td>
</tr>
<tr>
<td>waitTime</td>
<td>The time to wait on queues for messages of interest for the clients; in seconds.</td>
</tr>
</tbody>
</table>

getAgentList()

Retrieves the list of Agents for which the Listener provides its services.

**Syntax**

```cpp
vector<Agent> getAgentList() const;
```
**getTimeoutForListen()**

Retrieves the time out for a call.

**Syntax**

```cpp
int getTimeOutForListen() const;
```

**listen()**

Listens for Messages on behalf of specified Agents for the amount of time specified by a previous `setTimeOutForListen()` call. Returns the Agent for which there is a Message.

Note that this is a blocking call. Before this call, complete the following steps:

- Registers each Agent listener through a `setAgentList()` call.
- Make a blocking call to `setTimeOutForListen()` that returns when a Message for one of the Agents on the list arrives. If no Messages arrive before the wait time expires, the call returns an error.

**Syntax**

```cpp
Agent listen();
```

**setAgentList()**

Specifies the list of Agents for which the Listener provides its services.

**Syntax**

```cpp
void setAgentList(
    vector<Agent>& agList);  
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agList</td>
<td>The list of Agents.</td>
</tr>
</tbody>
</table>

**setTimeOutForListen()**

Specifies the time out for a `listen()` call.

**Syntax**

```cpp
void setTimeOutForListen(
    int waitTime);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>waitTime</td>
<td>The time interval, in seconds, during which the Listener is waiting for Messages at specified queues.</td>
</tr>
</tbody>
</table>
Map Class

The Map class is used to store the mapping of the SQL structured type to C++ classes. For each user defined type, the Object Type Translator (OTT) generates a C++ class declaration and implements the static methods `readSQL()` and `writeSQL()`. The `readSQL()` method is called when the object from the server appears in the application as a C++ class instance. The `writeSQL()` method is called to marshal the object in the application cache to server data when it is being written/flushed to the server. The `readSQL()` and `writeSQL()` methods generated by OTT are based upon the OCCI standard C++ mappings.

If you want to override the standard OTT generated mappings with customized mappings, you must implement a custom C++ class along with the `readSQL()` and `writeSQL()` methods for each SQL structured type you must customize. In addition, you must add an entry for each such class into the Map member of the Environment.

**Table 13–24  Summary of Map Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put()</code></td>
<td>Add a map entry for the type <code>type_name</code>.</td>
</tr>
</tbody>
</table>

**put()**

Adds a map entry for the type, `type_name`, that must be customized; you must implement the `type_name` C++ class.

You must then add this information into a map object, which should be registered with the connection if the user wants the standard mappings to overridden. This registration can be done by calling the `put()` method after the environment is created passing the environment.

**Syntax**

```cpp
void put(
    const string &schemaType,
    void *(*rSQL)(void *),
    void (*wSQL)(void *, void *));
```

Registers a type and its corresponding C++ `readSQL` and `writeSQL` functions.

```cpp
void put(
    const string & schName,
    const string & typName,
    void *(*rSQL)(void *),
    void (*wSQL)(void *, void *));
```

Registers a type and its corresponding C++ `readSQL` and `writeSQL` functions; multibyte support.

```cpp
void put(
    const UString & schName,
    const UString & typName,
    void *(*rSQL)(void *),
    void (*wSQL)(void *, void *));
```

Registers a type and its corresponding C++ `readSQL` and `writeSQL` functions; unicode support.

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schemaType</td>
<td>The schema and typename, separated by &quot;.&quot;, like <code>HR.TYPE1</code></td>
</tr>
<tr>
<td>schName</td>
<td>Name of the schema</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>typName</td>
<td>Name of the type</td>
</tr>
<tr>
<td>rDQL</td>
<td>The readSQL function pointer of the C++ class that corresponds to the type</td>
</tr>
<tr>
<td>wSQL</td>
<td>The writeSQL function pointer of the C++ class that corresponds to the type</td>
</tr>
</tbody>
</table>
Message Class

A message is the unit that is enqueued dequeued. A Message object holds both its content, or payload, and its properties. This class provides methods to get and set message properties.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageState</td>
<td>■ MSG_WAITING indicates that the message delay time has not been reached</td>
</tr>
<tr>
<td></td>
<td>■ MSG_READY indicates that the message is ready to be processed</td>
</tr>
<tr>
<td></td>
<td>■ MSG_PROCESSED indicates that the message has been processed, and is being retained</td>
</tr>
<tr>
<td></td>
<td>■ MSG_EXPIRED indicates that the message has been moved to the exception queue.</td>
</tr>
<tr>
<td>PayloadType</td>
<td>■ RAW</td>
</tr>
<tr>
<td></td>
<td>■ ANYDATA</td>
</tr>
<tr>
<td></td>
<td>■ OBJECT</td>
</tr>
</tbody>
</table>

Table 13–26  Summary of Message Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>message() on page 13-121</td>
<td>Message class constructor.</td>
</tr>
<tr>
<td>getAnyData() on page 13-121</td>
<td>Retrieves AnyData payload of the message.</td>
</tr>
<tr>
<td>getAttemptsToDequeue() on page 13-122</td>
<td>Retrieves the number of attempts made to dequeue the message.</td>
</tr>
<tr>
<td>getBytes() on page 13-122</td>
<td>Retrieves Bytes payload of the message.</td>
</tr>
<tr>
<td>getCorrelationId() on page 13-122</td>
<td>Retrieves the identification string.</td>
</tr>
<tr>
<td>getDelay() on page 13-122</td>
<td>Retrieves delay with which message was enqueued.</td>
</tr>
<tr>
<td>getExceptionQueueName() on page 13-122</td>
<td>Retrieves name of queue to which Message is moved when it cannot be processed.</td>
</tr>
<tr>
<td>getExpiration() on page 13-122</td>
<td>Retrieves the expiration of the message.</td>
</tr>
<tr>
<td>getMessageEnqueuedTime() on page 13-123</td>
<td>Retrieves time at which message was enqueued.</td>
</tr>
<tr>
<td>getMessageState() on page 13-123</td>
<td>Retrieves state of the message at time of enqueuing.</td>
</tr>
<tr>
<td>getObject() on page 13-123</td>
<td>Retrieves object payload of the message.</td>
</tr>
<tr>
<td>getOriginalMessageId() on page 13-123</td>
<td>Retrieves the Id of the message that generated this message on the last queue.</td>
</tr>
<tr>
<td>getPayloadType() on page 13-123</td>
<td>Retrieves the type of the payload.</td>
</tr>
<tr>
<td>getPriority() on page 13-123</td>
<td>Retrieves the priority of the message.</td>
</tr>
<tr>
<td>getSenderId() on page 13-124</td>
<td>Retrieves the agent who send the Message.</td>
</tr>
<tr>
<td>isNull() on page 13-124</td>
<td>Tests whether the Message object is NULL.</td>
</tr>
</tbody>
</table>
Message Class

**Message()**

Message class constructor.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message(const Environment *env);</td>
<td>Creates a Message object within the specified Environment.</td>
</tr>
<tr>
<td>Message(const Message&amp; mes);</td>
<td>Copy constructor.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>The environment of the Message.</td>
</tr>
<tr>
<td>mes</td>
<td>The original Message.</td>
</tr>
</tbody>
</table>

**getAnyData()**

Retrieves the AnyData payload of the Message.

**Syntax**

AnyData getAnyData() const;
getAttemptsToDequeue()

Retrieves the number of attempts made to dequeue the message. This property cannot be retrieved while enqueuing.

**Syntax**
```cpp
int getAttemptsToDequeue() const;
```

getBytes()

Retrieves Bytes payload of the Message.

**Syntax**
```cpp
Bytes getBytes() const;
```

defCorrelationId()

Retrieves the identification string.

**Syntax**
```cpp
string getCorrelationId() const;
```

defDelay()

Retrieves the delay (in seconds) with which the Message was enqueued.

**Syntax**
```cpp
int getDelay() const;
```

defExceptionQueueName()

Retrieves the name of the queue to which the Message is moved, in cases when the Message cannot be processed successfully.

**Syntax**
```cpp
string getExceptionQueueName() const;
```

defExpiration()

Retrieves the expiration time of the Message (in seconds). This is the duration for which the message is available for dequeuing.

**Syntax**
```cpp
int getExpiration() const;
```
getMessageEnqueuedTime()

Retrieves the time at which the message was enqueued, in Date format. This value is determined by the system, and cannot be set by the user.

**Syntax**

```
Date getMessageEnqueuedTime() const;
```

getMessageState()

Retrieves the state of the message at the time of enqueuing. This parameter cannot be set at enqueuing time. MessageState is defined in Table 13–25 on page 13-120.

**Syntax**

```
MessageState getMessageState() const;
```

getObject()

Retrieves object payload of the Message.

**Syntax**

```
PObject* getObject();
```

getOriginalMessageId()

Retrieves the original message Id. When a message is propagated from one queue to another, gets the ID to the last queue that generated this message.

**Syntax**

```
Bytes getOriginalMessageId() const;
```

getPayloadType()

Retrieves the type of the payload, as defined for PayloadType in Table 13–25 on page 13-120.

**Syntax**

```
PayloadType getPayloadType() const;
```

g GettyPriority()

Retrieves the priority of the Message.

**Syntax**

```
int getPriority() const;
```
getSenderId()

Retrieves the agent who send the Message.

Syntax
Agent getSenderId() const;

isNull()

Tests whether the Message object is NULL. If the Message object is NULL, then TRUE is returned; otherwise, FALSE is returned.

Syntax
bool isNull() const;

operator=(

Assignment operator for Message.
void operator=(
    const Message& mes);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mes</td>
<td>Original message.</td>
</tr>
</tbody>
</table>

setAnyData()

Specifies AnyData payload of the Message.

Syntax
void setAnyData(
    const AnyData& anydata);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anydata</td>
<td>Data content of the Message.</td>
</tr>
</tbody>
</table>

setBytes()

Specifies Bytes payload of the Message.

Syntax
void setBytes(
    const Bytes& bytes);
setCorrelationId()

Specifies the identification string. This parameter is set at enqueuing time by the Producer. Messages can be dequeued with this id. The id can contain wildcard characters.

**Syntax**

```cpp
void setCorrelationId(
    const string& id);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The id; upper limit of 128 bytes.</td>
</tr>
</tbody>
</table>

setDelay()

Specifies the time (in seconds) to delay the enqueued Message. After the delay ends, the Message is available for dequeuing.

Note that dequeuing by msgid overrides the delay specification. A Message enqueued with delay is in the WAITING state. Delay is set by the producer of the Message.

**Syntax**

```cpp
void setDelay(
    int delay);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay</td>
<td>The delay.</td>
</tr>
</tbody>
</table>

setExceptionQueueName()

Specifies the name of the queue to which the Message object is moved if it cannot be processed successfully. The queue name must be valid.

Note that if the exception queue does not exist at the time of the move, the Message is moved to the default exception queue associated with the queue table; a warning is logged in the alert log.

Also note that if the default exception queue is used, the parameter returns a NULL value at enqueuing time; the attribute must refer to a valid queue name.

**Syntax**

```cpp
void setExceptionQueueName(
    const string& queue);
```
setExpiration()

Specifies the duration time (in seconds) that the Message object is available for dequeuing. A Message expires after this time.

**Syntax**

```cpp
void setExpiration(
    int exp);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>The duration of expiration.</td>
</tr>
</tbody>
</table>

setNull()

Sets the Message object to NULL. Before the Connection is destroyed by the terminateConnection() call of the Environment Class, all Message objects must be set to NULL.

**Syntax**

```cpp
void setNull();
```

setObject()

Specifies object payload of the Message.

**Syntax**

```cpp
void setObject(
    PObject& pobj);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pobj</td>
<td>Content of the data</td>
</tr>
</tbody>
</table>

setOriginalMessageId()

Sets the Id of the last queue that generated the message, when a message is propagated from one queue to another.

**Syntax**

```cpp
void setOriginalMessageId(
    const Bytes& queue);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>The last queue.</td>
</tr>
</tbody>
</table>
setPriority()

Specifies the priority of the Message object. This property is set during enqueuing time, and can be negative. Default is 0.

**Syntax**

```java
void setPriority(
    int priority);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>The priority of the Message.</td>
</tr>
</tbody>
</table>

setRecipientList()

Specifies the list of Agents for whom the Message is intended. These recipients are not identical to subscribers of the queue. The property is set during enqueuing. All Agents in the list must be valid. The recipient list overrides the default subscriber list.

**Syntax**

```java
void setRecipientList(
    vector<Agent>& agentList);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agentList</td>
<td>The list of Agents.</td>
</tr>
</tbody>
</table>

setSenderId()

Specifies the sender of the Message.

**Syntax**

```java
void setSenderId(
    const Agent& sender);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sender</td>
<td>Sender id.</td>
</tr>
</tbody>
</table>
A MetaData object can be used to describe the types and properties of the columns in a ResultSet or the existing schema objects in the database. It also provides information about the database as a whole. The enumerated values of MetaData are in Table 13–27, and the summary of its methods is in Table 13–28 on page 13-136.

**Table 13–27  Enumerated Values Used by MetaData Class**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParamType</td>
<td>The parameter types for objects are:</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_ARG is the argument of a function or procedure.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_COL is the column of a table or view.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_DATABASE is the database.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_FUNC is the function.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_PKG is the package.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_PROC is the procedure.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_SCHEMA is the schema.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_SEQ is the sequence.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_SYN is the synonym.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_TABLE is the table.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_TYPE is the type.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_TYPE_ARG is the argument of a type method.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_TYPE_ATTR is the attribute of a type.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_TYPE_COLL is the collection type information.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_TYPE_METHOD is the method of a type.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_TYPE_RESULT is the results of a method.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_UNK is the object of an unknown type.</td>
</tr>
<tr>
<td></td>
<td>■ PTYPE_VIEW is the view.</td>
</tr>
<tr>
<td>AttrId common to all parameters</td>
<td>Attributes of all parameters:</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_OBJ_ID is the object or schema id.</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_OBJ_NAME is either the database name, or the object name in a schema.</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_OBJ_SCHEMA is the name of the schema describing the object.</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_PTYPE is the type of information described by a parameter, PType</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_TIMESTAMP is the timestamp of an object.</td>
</tr>
</tbody>
</table>
Parameters for a table or view (ParamType of PTYPE_TABLE and PTYPE_VIEW) have the following type-specific attributes:

- **ATTR_OBJID** is the object id
- **ATTR_NUM_COLS** is the number of columns
- **ATTR_LIST_COLUMNS** is the column list
- **ATTR_REF_TDO** is the REF to the TDO of the base type in case of extent tables
- **ATTR_IS_TEMPORARY** indicates the table is temporary
- **ATTR_IS_TYPED** indicates the table is typed
- **ATTR_DURATION** is the duration of a temporary table. Values can be DURATION_SESSION, DURATION_TRANS, and DURATION_NULL, as defined for attribute AttrValues

Parameters for a table or view (ParamType of PTYPE_TABLE) in summary:

- **ATTR_RDBA** indicates the data block address of the segment header
- **ATTR_TABLESPACE** indicates the tablespace the table resides in
- **ATTR_CLUSTERED** indicates the table is clustered
- **ATTR_PARTITIONED** indicates the table is partitioned
- **ATTR_INDEX_ONLY** indicates the table is index-only

Parameters for functions and procedures (ParamType of PTYPE_FUNC and PTYPE_PROC, respectively) in summary:

- **ATTR_LIST_ARGUMENTS** indicates the argument list
- **ATTR_IS_INVOKER_RIGHTS** indicates the procedure or function has invoker's rights
- **ATTR_NAME** indicates the name of the procedure or function
- **ATTR_OVERLOAD_ID** indicates the overloading ID number, relevant when the procedure or function is part of a class and it is overloaded; values returned may be different from direct query of a PL/SQL function or procedure

Parameters for functions and procedures (ParamType of PTYPE_FUNC and PTYPE_PROC, respectively) in summary:

- **ATTR_LIST_SUBPROGRAMS** indicates the subprogram list
- **ATTR_IS_INVOKER_RIGHTS** indicates the procedure or function has invoker's rights

Parameters for functions and procedures (ParamType of PTYPE_PKG, respectively) in summary:

- **ATTR_LIST_SUBPROGRAMS** indicates the subprogram list
- **ATTR_IS_INVOKER_RIGHTS** indicates the procedure or function has invoker's rights

---

**Table 13–27 (Cont.) Enumerated Values Used by MetaData Class**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AttrId for Tables and Views</strong></td>
<td>Parameters for a table or view (ParamType of PTYPE_TABLE and PTYPE_VIEW) have the following type-specific attributes:</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_OBJID</strong> is the object id</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_NUM_COLS</strong> is the number of columns</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_LIST_COLUMNS</strong> is the column list</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_REF_TDO</strong> is the REF to the TDO of the base type in case of extent tables</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_IS_TEMPORARY</strong> indicates the table is temporary</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_IS_TYPED</strong> indicates the table is typed</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_DURATION</strong> is the duration of a temporary table. Values can be DURATION_SESSION, DURATION_TRANS, and DURATION_NULL, as defined for attribute AttrValues</td>
</tr>
<tr>
<td><strong>AttrId for Tables only</strong></td>
<td>Parameters for a table only (ParamType of PTYPE_TABLE):</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_RDBA</strong> indicates the data block address of the segment header</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_TABLESPACE</strong> indicates the tablespace the table resides in</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_CLUSTERED</strong> indicates the table is clustered</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_PARTITIONED</strong> indicates the table is partitioned</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_INDEX_ONLY</strong> indicates the table is index-only</td>
</tr>
<tr>
<td><strong>AttrId for Functions and Procedures</strong></td>
<td>Parameters for functions and procedures (ParamType of PTYPE_FUNC and PTYPE_PROC, respectively):</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_LIST_ARGUMENTS</strong> indicates the argument list</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_IS_INVOKER_RIGHTS</strong> indicates the procedure or function has invoker's rights</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_NAME</strong> indicates the name of the procedure or function</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_OVERLOAD_ID</strong> indicates the overloading ID number, relevant when the procedure or function is part of a class and it is overloaded; values returned may be different from direct query of a PL/SQL function or procedure</td>
</tr>
<tr>
<td><strong>AttrId for Packages</strong></td>
<td>Parameters for packages (ParamType of PTYPE_PKG):</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_LIST_SUBPROGRAMS</strong> indicates the subprogram list</td>
</tr>
<tr>
<td></td>
<td>- <strong>ATTR_IS_INVOKER_RIGHTS</strong> indicates the procedure or function has invoker's rights</td>
</tr>
</tbody>
</table>
Table 13–27 (Cont.) Enumerated Values Used by MetaData Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttrId for Types</td>
<td>Parameter is for types (PTYPE_TYPE):</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_REF_TDO indicates the in-memory REF of the type descriptor for the type, if the column type is an object type. If space has not been reserved, then it is allocated implicitly in the cache. The caller can then pin the object.</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_TYPECODE indicates the data type code</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_COLLECTION_TYPECODE indicates the typecode of collection, if type is collection</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_INCOMPLETE_TYPE indicates that this is an incomplete type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_SYSTEM_TYPE indicates that this is a system generated type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_PREDEFINED_TYPE indicates that this is a predefined type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_TRANSIENT_TYPE indicates that this is a transient type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_SYSTEM_GENERATED_TYPE indicates that this is a system generated type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_HAS_NESTED_TABLE indicates that this type contains a nested table attribute</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_HAS_LOB indicates that this type contains a LOB attribute</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_HAS_FILE indicates that this type contains a BFILE attribute</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_COLLECTION_ELEMENT indicates a reference to a collection element</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_NUM_TYPE_ATTRS indicates the number of type attributes</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_LIST_TYPE_ATTRS indicates the list of type attributes</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_NUM_TYPE_METHODS indicates the number of type methods</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_LIST_TYPE_METHODS indicates the list of type methods</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_MAP_METHOD indicates the map method of the type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_ORDER_METHOD indicates the order method of the type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_INVOKER_RIGHTS indicates the type has invoker’s rights</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_NAME indicates the type attribute name</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_SCHEMA_NAME indicates the schema where the type is created</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_FINAL_TYPE indicates this is a final type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_INSTANTIABLE_TYPE indicates this is an instantiable type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_SUBTYPE indicates this is a subtype</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_SUPERTYPE_SCHEMA_NAME indicates the name of the schema that contains the supertype</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_SUPERTYPE_NAME indicates the name of the supertype</td>
</tr>
</tbody>
</table>
Table 13-27 (Cont.) Enumerated Values Used by MetaData Class

<table>
<thead>
<tr>
<th>Attribute for Type Attributes</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttrId for Type Attributes</td>
<td>Parameter is for attributes of types (ParamType of PTYPE_TYPE_ATTR):</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_DATA_SIZE indicates the maximum size of the type attribute</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_TYPECODE indicates the data type code</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_DATA_TYPE indicates the data type of the type attribute</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_NAME indicates the name of the procedure or function</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_PRECISION indicates the precision of numeric type attributes.</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_SCALE indicates the scale of the numeric type attributes</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_TYPE_NAME indicates a type name</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_SCHEMA_NAME indicates the name of the schema where the type has been created</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_REF_TDO indicates the in-memory REF of the type, if the column type is an object type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_CHARSET_ID indicates the characterset ID</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_CHARSET_FORM indicates the characterset form</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_FSPRECISION indicates the fractional seconds precision of a Timestamp, IntervalDS or IntervalYM</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_LFPRECISION indicates the leading field precision of an IntervalDS or IntervalYM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AttrId for Type Methods</th>
<th>Parameter is for methods of types (ParamType of PTYPE_TYPE_METHOD):</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>■ ATTR_NAME indicates the name of the procedure or function</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_ENCAPSULATION indicates the method’s level of encapsulation</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_LIST_ARGUMENTS indicates the argument list</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_CONSTRUCTOR indicates the method is a constructor</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_DESTRUCTOR indicates the method is a destructor</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_OPERATOR indicates the method is an operator</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_SELFISH indicates the method is selfish</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_MAP indicates the method is a map method</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_ORDER indicates the method is an order method</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_RNDS indicates that the method is in “read no data” state</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_RNPS indicates that the method is in “read no process” state</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_WNDS indicates that the method is in “write no data” state</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_WNPS indicates that the method is in “write no process” state</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_FINAL_METHOD indicates that this is a final method</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_INSTANTIABLE_METHOD indicates that this is an instantiable method</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_OVERRIDING_METHOD indicates that this is an overriding method</td>
</tr>
</tbody>
</table>
### Table 13–27 (Cont.) Enumerated Values Used by MetaData Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttrId for Collections</td>
<td>Parameter is for collections (ParamType of PTYPE_TYPE_COLL):</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_DATA_SIZE indicates</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_TYPECODE indicates</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_DATA_TYPE indicates the data type of the type attribute</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_NUM_ELEMS indicates the number of elements in a collection</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_NAME indicates the name of the type attribute</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_PRECISION indicates the precision of a numeric attribute</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_SCALE indicates the scale of a numeric attribute</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_TYPE_NAME indicates the type name</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_SCHEMA_NAME indicates the schema where the type has been created</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_REF_TDO indicates the in-memory REF of the type, if the column type is an object type. If the space has not been reserved, it is allocated implicitly in the cache. The caller can then pin the object.</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_CHARSET_ID indicates the characterset id</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_CHARSET_FORM indicates the characterset form</td>
</tr>
<tr>
<td>AttrId for Synonyms</td>
<td>Parameter is for synonyms (ParamType of PTYPE_SYN):</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_OBJID indicates the object id</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_SCHEMA_NAME indicates the schema name of the synonym translation</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_NAME indicates a NULL-terminated object name of the synonym translation</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_LINK indicates a NULL-terminated database link name of the synonym installation</td>
</tr>
<tr>
<td>AttrId for Sequences</td>
<td>Parameter is for sequences (ParamType of PTYPE_SEQ):</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_OBJID indicates the object id</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_MIN indicates the minimum value</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_MAX indicates the maximum value</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_INCR indicates the increment</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_CACHE indicates the number of sequence numbers cached; 0 if the sequence is not cached</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_ORDER indicates whether the sequence is ordered</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_HW_MARK indicates the &quot;high-water mark&quot;</td>
</tr>
</tbody>
</table>
**MetaData Class**

**Table 13–27 (Cont.) Enumerated Values Used by MetaData Class**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttrId for Columns</td>
<td>Parameter is for columns of tables or views (ParamType of PTYPE_COL):</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_CHAR_USED indicates the type of length semantics of the column. 0 means byte-length semantics and 1 means character-length semantics.</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_CHAR_SIZE indicates the column character length, or number of characters allowed in a column</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_DATA_SIZE indicates the maximum size of a column, or number of bytes allowed in a column</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_DATA_TYPE indicates the data type of the column</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_NAME indicates the column name</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_PRECISION indicates the precision of numeric columns</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_SCALE indicates the scale of numeric columns</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_IS_NULL indicates 0 if NULL values are not permitted for the column</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_TYPE_NAME indicates a type name</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_SCHEMA_NAME indicates the schema where the type was created</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_REF_TDO indicates the REF for the type, if the column is of object type</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_CHARSET_ID indicates the characterset ID</td>
</tr>
<tr>
<td></td>
<td>■ ATTR_CHARSET_FORM indicates the characterset form</td>
</tr>
</tbody>
</table>
### Table 13–27  (Cont.) Enumerated Values Used by MetaData Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AttrId for Arguments and Results</strong></td>
<td>Parameter for arguments of a procedure or function (PTYPE_ARG), a method (PTYPE_TYPE_ARG), or a result (PTYPE_TYPE_RESULT)</td>
</tr>
<tr>
<td></td>
<td>- ATTR_NAME indicates the argument name</td>
</tr>
<tr>
<td></td>
<td>- ATTR_POSITION indicates the position of the argument in the list</td>
</tr>
<tr>
<td></td>
<td>- ATTR_TYPECODE indicates the typecode</td>
</tr>
<tr>
<td></td>
<td>- ATTR_DATA_TYPE indicates the data type</td>
</tr>
<tr>
<td></td>
<td>- ATTR_DATA_SIZE indicates the size of the data type</td>
</tr>
<tr>
<td></td>
<td>- ATTR_PRECISION indicates the precision of a numeric argument</td>
</tr>
<tr>
<td></td>
<td>- ATTR_SCALE indicates the scale of a numeric argument</td>
</tr>
<tr>
<td></td>
<td>- ATTR_LEVEL indicates the data type level</td>
</tr>
<tr>
<td></td>
<td>- ATTR_HAS_DEFAULT indicates whether an argument has a default</td>
</tr>
<tr>
<td></td>
<td>- ATTR_LIST_ARGUMENTS indicates the list of arguments at the next level, for records or table types</td>
</tr>
<tr>
<td></td>
<td>- ATTR_IOMODE indicates the argument mode: 0 for IN, 1 for OUT, 2 for IN/OUT</td>
</tr>
<tr>
<td></td>
<td>- ATTR_RADIX indicates the radix of a number type</td>
</tr>
<tr>
<td></td>
<td>- ATTR_IS_NULL indicates 0 if NULL values are not permitted</td>
</tr>
<tr>
<td></td>
<td>- ATTR_TYPE_NAME indicates the type name</td>
</tr>
<tr>
<td></td>
<td>- ATTR_SCHEMA_NAME indicates the schema name where the type was created</td>
</tr>
<tr>
<td></td>
<td>- ATTR_SUB_NAME indicates the type name for package local types</td>
</tr>
<tr>
<td></td>
<td>- ATTR_LINK indicates a NULL-terminated database link name where the type is defined, for package local types when the package is remote</td>
</tr>
<tr>
<td></td>
<td>- ATTR_REF_TDO is the REF to the TDO of the type if the argument is an object</td>
</tr>
<tr>
<td></td>
<td>- ATTR_CHARSET_ID indicates the characterset ID</td>
</tr>
<tr>
<td></td>
<td>- ATTR_CHARSET_FORM indicates the characterset form</td>
</tr>
<tr>
<td><strong>AttrId for Schemas</strong></td>
<td>Parameter is for schemas (ParamType of PTYPE_SCHEMA):</td>
</tr>
<tr>
<td></td>
<td>- ATTR_LIST_OBJECTS indicates the list of objects in the schema</td>
</tr>
<tr>
<td><strong>AttrId for Lists</strong></td>
<td>Parameter is for list of columns, arguments or subprograms:</td>
</tr>
<tr>
<td></td>
<td>- ATTR_LIST_COLUMNS indicates a column list</td>
</tr>
<tr>
<td></td>
<td>- ATTR_LIST_ARGUMENTS indicates a procedure or function argument list</td>
</tr>
<tr>
<td></td>
<td>- ATTR_LIST_SUBPROGRAMS indicates a subprogram list</td>
</tr>
<tr>
<td></td>
<td>- ATTR_LIST_TYPE_ATTRIBS indicates a type attribute list</td>
</tr>
<tr>
<td></td>
<td>- ATTR_TYPE_METHODS indicates a type method list</td>
</tr>
<tr>
<td></td>
<td>- ATTR_TYPE_OBJECTS indicates a list of objects in a schema</td>
</tr>
<tr>
<td></td>
<td>- ATTR_LIST_SCHEMAS indicates a list of schemas in a database</td>
</tr>
<tr>
<td>Attribute for Databases</td>
<td>Options</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>ATTR_VERSION indicates the database version</td>
<td></td>
</tr>
<tr>
<td>ATTR_CHARSET_ID indicates the characterset ID of the database</td>
<td></td>
</tr>
<tr>
<td>ATTR_NCHARSET_ID indicates the national characterset of the database</td>
<td></td>
</tr>
<tr>
<td>ATTR_LIST_SCHEMAS indicates the list of schemas, PTYPE_SCHEMA</td>
<td></td>
</tr>
<tr>
<td>ATTR_MAX_PROC_LEN indicates the maximum length of a procedure name</td>
<td></td>
</tr>
<tr>
<td>ATTR_MAX_COLUMN_LEN indicates the maximum length of a column name</td>
<td></td>
</tr>
<tr>
<td>ATTR_CURSOR_COMMIT_BEHAVIOR indicates how a commit affects cursors and prepared statements. Values can be CURSOR_OPEN and CURSOR_CLOSED, as defined for attribute AttrValues</td>
<td></td>
</tr>
<tr>
<td>ATTR_MAX_CATALOG_NAMELEN indicates the maximum length of a database (catalog) name</td>
<td></td>
</tr>
<tr>
<td>ATTR_CATALOG_LOCATION indicates the position of the catalog in a qualified table. Values can be CL_START and CL_END, as defined for attribute AttrValues</td>
<td></td>
</tr>
<tr>
<td>ATTR_SAVEPOINT_SUPPORT indicates whether the database supports savepoints. Values can be SP_SUPPORTED and SP_UNSUPPORTED, as defined for attribute AttrValues</td>
<td></td>
</tr>
<tr>
<td>ATTR_NOWAIT_SUPPORT indicates whether the database supports the &quot;no wait&quot; condition. Values can be NW_SUPPORTED and NW_UNSUPPORTED, as defined for attribute AttrValues</td>
<td></td>
</tr>
<tr>
<td>ATTR_AUTO_COMMIT_DDL indicates if an autocommit mode is required for DDL statements. Values can be AC_DDL and NO_AC_DDL, as defined for attribute AttrValues</td>
<td></td>
</tr>
<tr>
<td>ATTR_LOCKING_MODE indicates the locking mode for the database. Values can be LOCK_IMMEDIATE and LOCK_DELAYED, as defined for attribute AttrValues</td>
<td></td>
</tr>
</tbody>
</table>
Table 13–27  (Cont.) Enumerated Values Used by MetaData Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttrValues</td>
<td>Attribute values are returned on executing a get xxx() method and</td>
</tr>
<tr>
<td></td>
<td>passing in an attribute, for which these are the results:</td>
</tr>
<tr>
<td></td>
<td>■ DURATION_SESSION is the duration of a temporary table: session.</td>
</tr>
<tr>
<td></td>
<td>■ DURATION_TRANS is the duration of a temporary table: transaction.</td>
</tr>
<tr>
<td></td>
<td>■ DURATION_NULL is the duration of a temporary table: table not</td>
</tr>
<tr>
<td></td>
<td>temporary.</td>
</tr>
<tr>
<td></td>
<td>■ TYPEENCAP_PRIVATE is the encapsulation level of the method:</td>
</tr>
<tr>
<td></td>
<td>private.</td>
</tr>
<tr>
<td></td>
<td>■ TYPEENCAP_PUBLIC is the encapsulation level of the method:</td>
</tr>
<tr>
<td></td>
<td>public.</td>
</tr>
<tr>
<td></td>
<td>■ TYPEPARAM_IN is the argument mode: IN.</td>
</tr>
<tr>
<td></td>
<td>■ TYPEPARAM_OUT is the argument mode: OUT.</td>
</tr>
<tr>
<td></td>
<td>■ TYPEPARAM_INOUT is the argument mode: IN/OUT.</td>
</tr>
<tr>
<td></td>
<td>■ CURSOR_OPEN is the effect of COMMIT operation on cursors and</td>
</tr>
<tr>
<td></td>
<td>prepared statements in the database: preserve cursor state as</td>
</tr>
<tr>
<td></td>
<td>before the COMMIT operation.</td>
</tr>
<tr>
<td></td>
<td>■ CURSOR_CLOSED is the effect of COMMIT operation on cursors and</td>
</tr>
<tr>
<td></td>
<td>prepared statements in the database: cursors are closed on COMMIT,</td>
</tr>
<tr>
<td></td>
<td>but the application can still rerun the statement without</td>
</tr>
<tr>
<td></td>
<td>preparing it again.</td>
</tr>
<tr>
<td></td>
<td>■ CL_START is the position of the catalog in a qualified table:</td>
</tr>
<tr>
<td></td>
<td>start.</td>
</tr>
<tr>
<td></td>
<td>■ CL_END is the position of the catalog in a qualified table: end.</td>
</tr>
<tr>
<td></td>
<td>■ SP_SUPPORTED is the database supports savepoints.</td>
</tr>
<tr>
<td></td>
<td>■ SP_UNSUPPORTED is the database does not support savepoints.</td>
</tr>
<tr>
<td></td>
<td>■ NW_SUPPORTED is the database supports nowait clause.</td>
</tr>
<tr>
<td></td>
<td>■ NW_UNSUPPORTED is the database does not supports nowait clause.</td>
</tr>
<tr>
<td></td>
<td>■ AC_DDL is the autocommit mode required for DDL statements.</td>
</tr>
<tr>
<td></td>
<td>■ NO_AC_DDL is the autocommit mode not required for DDL</td>
</tr>
<tr>
<td></td>
<td>statements.</td>
</tr>
<tr>
<td></td>
<td>■ LOCK_IMMEDIATE is the locking mode for the database: immediate.</td>
</tr>
<tr>
<td></td>
<td>■ LOCK_DELAYED is the locking mode for the database: delayed.</td>
</tr>
</tbody>
</table>

Table 13–28  Summary of MetaData Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MetaData() on page 13-137</td>
<td>MetaData class constructor.</td>
</tr>
<tr>
<td>getAttributeCount() on page 13-137</td>
<td>Gets the count of the attribute as a MetaData object</td>
</tr>
<tr>
<td>getAttributeId() on page 13-137</td>
<td>Gets the ID of the specified attribute</td>
</tr>
<tr>
<td>getAttributeType() on page 13-137</td>
<td>Gets the type of the specified attribute</td>
</tr>
<tr>
<td>getBoolean() on page 13-138</td>
<td>Gets the value of the attribute as a C++ boolean</td>
</tr>
<tr>
<td>getInt() on page 13-138</td>
<td>Gets the value of the attribute as a C++ int.</td>
</tr>
<tr>
<td>getMetaData() on page 13-138</td>
<td>Gets the value of the attribute as a MetaData object</td>
</tr>
<tr>
<td>getNumber() on page 13-139</td>
<td>Returns the specified attribute as a Number object.</td>
</tr>
<tr>
<td>getRef() on page 13-139</td>
<td>Gets the value of the attribute as a Ref&lt;T&gt;.</td>
</tr>
</tbody>
</table>
Table 13–28  (Cont.) Summary of MetaData Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getString() on page 13-139</td>
<td>Gets the value of the attribute as a string.</td>
</tr>
<tr>
<td>getTimeStamp() on page 13-139</td>
<td>Gets the value of the attribute as a Timestamp object</td>
</tr>
<tr>
<td>getUInt() on page 13-140</td>
<td>Gets the value of the attribute as a C++ unsigned int.</td>
</tr>
<tr>
<td>getUString() on page 13-140</td>
<td>Returns the value of the attribute as a UString in the character set associated with the metadata.</td>
</tr>
<tr>
<td>getVector() on page 13-140</td>
<td>Gets the value of the attribute as a C++ vector.</td>
</tr>
<tr>
<td>operator=() on page 13-140</td>
<td>Assigns one metadata object to another.</td>
</tr>
</tbody>
</table>

**MetaData()**

MetaData class constructor.

**Syntax**

```cpp
MetaData{
    const MetaData &cmd);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>The source that the MetaData object is copied from.</td>
</tr>
</tbody>
</table>

**getAttributeCount()**

This method returns the number of attributes related to the metadata object.

**Syntax**

```cpp
unsigned int getAttributeCount() const;
```

**getAttributeId()**

This method returns the attribute ID, such as ATTR_NUM_COLS, of the attribute represented by the attribute number specified.

**Syntax**

```cpp
AttrId getAttributeId(
    unsigned int attributeNum) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeNum</td>
<td>The number of the attribute for which the attribute ID is to be returned.</td>
</tr>
</tbody>
</table>

**getAttributeType()**

This method returns the attribute type, such as NUMBER or INT, of the attribute represented by attribute number specified.
getBoolean()

Syntax

Type getAttributeType(
    unsigned int attributeNum) const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeNum</td>
<td>The number of the attribute for which the attribute type is to be returned.</td>
</tr>
</tbody>
</table>

getBoolean()

This method returns the value of the attribute as a C++ boolean. If the value is a SQL NULL, the result is FALSE.

Syntax

bool getBoolean(
    MetaData::AttrId attributeId) const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeId</td>
<td>The attribute ID</td>
</tr>
</tbody>
</table>

getInt()

This method returns the value of the attribute as a C++ int. If the value is SQL NULL, the result is 0.

Syntax

int getInt(
    MetaData::AttrId attributeId) const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeId</td>
<td>The attribute ID</td>
</tr>
</tbody>
</table>

getMetaData()

This method returns a MetaData instance holding the attribute value. A metadata attribute value can be retrieved as a MetaData instance. This method can only be called on attributes of the metadata type.

Syntax

MetaData getMetaData(
    MetaData::AttrId attributeId) const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeId</td>
<td>The attribute ID</td>
</tr>
</tbody>
</table>
**getNumber()**

This method returns the value of the attribute as a *Number* object. If the value is a SQL NULL, the result is NULL.

**Syntax**

```cpp
Number getNumber(
    Metadata::AttrId attributeId) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeId</td>
<td>The attribute ID</td>
</tr>
</tbody>
</table>

**getRef()**

This method returns the value of the attribute as a *RefAny*, or Ref to a TDO. If the value is SQL NULL, the result is NULL.

**Syntax**

```cpp
RefAny getRef(
    Metadata::AttrId attributeId) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeId</td>
<td>The attribute ID</td>
</tr>
</tbody>
</table>

**getString()**

This method returns the value of the attribute as a string. If the value is SQL NULL, the result is NULL.

**Syntax**

```cpp
string getString(
    Metadata::AttrId attributeId) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeId</td>
<td>The attribute ID</td>
</tr>
</tbody>
</table>

**getTimeStamp()**

This method returns the value of the attribute as a *Timestamp* object. If the value is a SQL NULL, the result is NULL.

**Syntax**

```cpp
Timestamp getTimeStamp(
    Metadata::AttrId attributeId) const;
```
getUInt()

This method returns the value of the attribute as a C++ unsigned `int`. If the value is a SQL NULL, the result is 0.

**Syntax**

```cpp
unsigned int getUInt(
    MetaData::AttrId attributeId) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeId</td>
<td>The attribute ID</td>
</tr>
</tbody>
</table>

getUString()

Returns the value of an attribute as a `UString` in the character set associated with the metadata.

**Syntax**

```cpp
UString getUString(
    MetaData::AttrId attributeId) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeId</td>
<td>The attribute ID</td>
</tr>
</tbody>
</table>

getVector()

This method returns a C++ vector containing the attribute value. A collection attribute value can be retrieved as a C++ vector instance. This method can only be called on attributes of a list type.

**Syntax**

```cpp
vector<MetaData> getVector(
    MetaData::AttrId attributeId) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributeId</td>
<td>The attribute ID</td>
</tr>
</tbody>
</table>

operator=()

This method assigns one `MetaData` object to another. This increments the reference count of the `MetaData` object that is assigned.
Syntax

```c++
void operator=(
    const MetaData &cmd);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>MetaData object to be assigned</td>
</tr>
</tbody>
</table>
NotifyResult Class

A NotifyResult object holds the notification information in the Streams AQ notification callback. It is created by OCCI before invoking a user-callback, and is destroyed after the user-callback returns.

Table 13—29 Summary of NotifyResult Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>getConsumerName()</td>
<td>Returns the name of the notification consumer.</td>
</tr>
<tr>
<td>getMessage()</td>
<td>Returns the message.</td>
</tr>
<tr>
<td>getMessageId()</td>
<td>Returns the message ID.</td>
</tr>
<tr>
<td>getPayload()</td>
<td>Returns the payload.</td>
</tr>
<tr>
<td>getQueueName()</td>
<td>Returns the name of the queue.</td>
</tr>
</tbody>
</table>

getConsumerName()

Gets the name of the consumer for which the message has been enqueued. In a single consumer queue, this is a empty string.

```cpp
string getConsumerName() const;
```

getAddress()

Gets the message which has been enqueued into the non-persistent queue.

```cpp
Message getMessage() const;
```

getMessageId()

Gets the id of the message which has been enqueued.

```cpp
Bytes getMessageId() const;
```

getPayload()

Gets the payload in case of a notification from NS_ANONYMOUS namespace.

```cpp
Bytes getPayload() const;
```

getQueueName()

Gets the name of the queue on which the enqueue has happened.

```cpp
string getQueueName() const;
```
Number Class

The Number class handles limited-precision signed base 10 numbers. A Number guarantees 38 decimal digits of precision. All positive numbers in the range displayed here can be represented to a full 38-digit precision:

\[ 10^{-130} \]

and

\[ 9.99999999999999999999999999999999999999\times10^{125} \]

The range of representable negative numbers is symmetrical.

The number zero can be represented exactly. Also, Oracle numbers have representations for positive and negative infinity. These are generally used to indicate overflow.

The internal storage type is opaque and private. Scale is not preserved when Number instances are created.

Number does not support the concept of NaN and is not IEEE-754-85 compliant. Number does support +Infinity and -Infinity.

Objects from the Number class can be used as standalone class objects in client side numeric computations. They can also be used to fetch from and set to the database.

Example 13–10  How to Retrieve and Use a Number Object

This example demonstrates a Number column value being retrieved from the database, a bind using a Number object, and a comparison using a standalone Number object.

```cpp
/* Create a connection */
Environment *env = Environment::createEnvironment(Environment::DEFAULT);
Connection *conn = Connection(user, passwd, db);

/* Create a statement and associate a select clause with it */
string sqlStmt = "SELECT department_id FROM DEPARTMENTS";
Statement *stmt = conn->createStatement(sqlStmt);

/* Run the statement to get a result set */
ResultSet *rset = stmt->executeQuery();
while(rset->next())
{
    Number deptId = rset->getNumber(1);
    /* Display the department id with the format string 9,999 */
    cout << 'Department Id' << deptId.toText(env, "9,999");

    /* Use the number obtained as a bind value in the following query */
    stmt->setSQL("SELECT * FROM EMPLOYEES WHERE department_id = :x");
    stmt->setNumber(1, deptId);
    ResultSet *rset2 = stmt->executeQuery();
    .
    .
}
/* Using a Number object as a standalone and the operations on them */

/* Create a number to a double value */
double value = 2345.123;
Number nul (value);
```
/ * Some common Number methods */
Number abs = nu1.abs();    /* absolute value */
Number sqrt = nu1.squareroot();    /* square root */
Environment *env = Environment::createEnvironment();

//create a null year-month interval
IntervalYM ym
if(ym.isNull())
    cout << "\n ym is null";

//assign a non null value to ym
IntervalYM anotherYM(env, "10-30");
ym = anotherYM;

//now all operations are valid on ym
int yr = ym.getYear();

## Table 13–30  Summary of Number Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number() on page 13-146</td>
<td>Number class constructor.</td>
</tr>
<tr>
<td>abs() on page 13-147</td>
<td>Returns the absolute value of the number.</td>
</tr>
<tr>
<td>arcCos() on page 13-147</td>
<td>Returns the arcCosine of the number.</td>
</tr>
<tr>
<td>arcSin() on page 13-147</td>
<td>Returns the arcSine of the number.</td>
</tr>
<tr>
<td>arcTan() on page 13-147</td>
<td>Returns the arcTangent of the number.</td>
</tr>
<tr>
<td>arcTan2() on page 13-148</td>
<td>Returns the arcTangent2 of the input number $y$ and this number $x$.</td>
</tr>
<tr>
<td>ceil() on page 13-148</td>
<td>Returns the smallest integral value not less than the value of the number.</td>
</tr>
<tr>
<td>cos() on page 13-148</td>
<td>Returns the cosine of the number.</td>
</tr>
<tr>
<td>exp() on page 13-148</td>
<td>Returns the natural exponent of the number.</td>
</tr>
<tr>
<td>floor() on page 13-148</td>
<td>Returns the largest integral value not greater than the value of the number.</td>
</tr>
<tr>
<td>fromBytes() on page 13-148</td>
<td>Returns a Number derived from a Bytes object.</td>
</tr>
<tr>
<td>fromText() on page 13-149</td>
<td>Returns a Number from a given number string, format string and NLS parameters specified.</td>
</tr>
<tr>
<td>hypCos() on page 13-149</td>
<td>Returns the hyperbolic cosine of the number.</td>
</tr>
<tr>
<td>hypSin() on page 13-150</td>
<td>Returns the hyperbolic sine of the number.</td>
</tr>
<tr>
<td>hypTan() on page 13-150</td>
<td>Returns the hyperbolic tangent of the number.</td>
</tr>
<tr>
<td>intPower() on page 13-150</td>
<td>Returns the number raised to the integer value specified.</td>
</tr>
<tr>
<td>isNull() on page 13-150</td>
<td>Checks if Number is NULL.</td>
</tr>
<tr>
<td>ln() on page 13-150</td>
<td>Returns the natural logarithm of the number.</td>
</tr>
<tr>
<td>log() on page 13-150</td>
<td>Returns the logarithm of the number to the base value specified.</td>
</tr>
<tr>
<td>operator++() on page 13-151</td>
<td>Increments the number.</td>
</tr>
<tr>
<td>operator--() on page 13-151</td>
<td>Decrements the number.</td>
</tr>
<tr>
<td>operator*() on page 13-101</td>
<td>Returns the product of two Numbers.</td>
</tr>
</tbody>
</table>
### Table 13–30 (Cont.) Summary of Number Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>operator/()</code> on page 13-152</td>
<td>Returns the quotient of two Numbers.</td>
</tr>
<tr>
<td><code>operator%()</code> on page 13-152</td>
<td>Returns the modulo of two Numbers.</td>
</tr>
<tr>
<td><code>operator+()</code> on page 13-152</td>
<td>Returns the sum of two Numbers.</td>
</tr>
<tr>
<td><code>operator-()</code> on page 13-152</td>
<td>Returns the negated value of Number.</td>
</tr>
<tr>
<td><code>operator-()</code> on page 13-152</td>
<td>Returns the difference between two Numbers.</td>
</tr>
<tr>
<td><code>operator&lt;()</code> on page 13-153</td>
<td>Checks if a number is less than an other number.</td>
</tr>
<tr>
<td><code>operator&lt;=()</code> on page 13-153</td>
<td>Checks if a number is less than or equal to an other number.</td>
</tr>
<tr>
<td><code>operator&gt;()</code> on page 13-154</td>
<td>Checks if a number is greater than an other number.</td>
</tr>
<tr>
<td><code>operator&gt;=()</code> on page 13-81</td>
<td>Checks if a number is greater than or equal to an other number.</td>
</tr>
<tr>
<td><code>operator=()</code> on page 13-80</td>
<td>Assigns one number to another.</td>
</tr>
<tr>
<td><code>operator==()</code> on page 13-80</td>
<td>Checks if two numbers are equal.</td>
</tr>
<tr>
<td><code>operator!=()</code> on page 13-80</td>
<td>Checks if two numbers are not equal.</td>
</tr>
<tr>
<td><code>operator*=()</code> on page 13-101</td>
<td>Multiplication assignment.</td>
</tr>
<tr>
<td><code>operator/=()</code> on page 13-103</td>
<td>Division assignment.</td>
</tr>
<tr>
<td><code>operator%=()</code> on page 13-156</td>
<td>Modulo assignment.</td>
</tr>
<tr>
<td><code>operator+=()</code> on page 13-156</td>
<td>Addition assignment.</td>
</tr>
<tr>
<td><code>operator-=()</code> on page 13-156</td>
<td>Subtraction assignment.</td>
</tr>
<tr>
<td><code>operator char()</code> on page 13-156</td>
<td>Returns Number converted to native char.</td>
</tr>
<tr>
<td><code>operator signed char()</code> on page 13-157</td>
<td>Returns Number converted to native signed char.</td>
</tr>
<tr>
<td><code>operator double()</code> on page 13-157</td>
<td>Returns Number converted to a native double.</td>
</tr>
<tr>
<td><code>operator float()</code> on page 13-157</td>
<td>Returns Number converted to a native float.</td>
</tr>
<tr>
<td><code>operator int()</code> on page 13-157</td>
<td>Returns Number converted to native integer.</td>
</tr>
<tr>
<td><code>operator long()</code> on page 13-157</td>
<td>Returns Number converted to native long.</td>
</tr>
<tr>
<td><code>operator long double()</code> on page 13-157</td>
<td>Returns Number converted to a native long double.</td>
</tr>
<tr>
<td><code>operator short()</code> on page 13-158</td>
<td>Returns Number converted to native short integer.</td>
</tr>
<tr>
<td><code>operator unsigned char()</code> on page 13-158</td>
<td>Returns Number converted to an unsigned native char.</td>
</tr>
<tr>
<td><code>operator unsigned int()</code> on page 13-158</td>
<td>Returns Number converted to an unsigned native integer.</td>
</tr>
<tr>
<td><code>operator unsigned long()</code> on page 13-158</td>
<td>Returns Number converted to an unsigned native long.</td>
</tr>
<tr>
<td><code>operator unsigned short()</code> on page 13-158</td>
<td>Returns Number converted to an unsigned native short integer.</td>
</tr>
<tr>
<td><code>power()</code> on page 13-158</td>
<td>Returns Number raised to the power of another number specified.</td>
</tr>
<tr>
<td><code>prec()</code> on page 13-159</td>
<td>Returns Number rounded to digits of precision specified.</td>
</tr>
</tbody>
</table>
Number()

Number class constructor.

### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number();</td>
<td>Default constructor.</td>
</tr>
<tr>
<td>Number( const Number &amp;srcNum);</td>
<td>Creates a copy of a Number.</td>
</tr>
<tr>
<td>Number( long double &amp;val);</td>
<td>Translates a native long double into a Number. The Number is created using the precision of the platform-specific constant LDBL_DIG.</td>
</tr>
<tr>
<td>Number( double val);</td>
<td>Translates a native double into a Number. The Number is created using the precision of the platform-specific constant DBL_DIG.</td>
</tr>
<tr>
<td>Number( float val);</td>
<td>Translates a native float into a Number. The Number is created using the precision of the platform-specific constant FLT_DIG.</td>
</tr>
<tr>
<td>Number( long val);</td>
<td>Translates a native long into a Number.</td>
</tr>
<tr>
<td>Number( int val);</td>
<td>Translates a native int into a Number.</td>
</tr>
<tr>
<td>Number( short val);</td>
<td>Translates a native short into a Number.</td>
</tr>
<tr>
<td>Number( char val);</td>
<td>Translates a native char into a Number.</td>
</tr>
<tr>
<td>Number( signed char val);</td>
<td>Translates a native signed char into a Number.</td>
</tr>
</tbody>
</table>

### Table 13–30 (Cont.) Summary of Number Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>round() on page 13-159</td>
<td>Returns Number rounded to decimal place specified. Negative values are allowed.</td>
</tr>
<tr>
<td>setNull() on page 13-159</td>
<td>Sets Number to NULL.</td>
</tr>
<tr>
<td>shift() on page 13-159</td>
<td>Returns a Number that is equivalent to the passed value * 10^n, where n may be positive or negative.</td>
</tr>
<tr>
<td>sign() on page 13-160</td>
<td>Returns the sign of the value of the passed value: -1 for the passed value &lt; 0, 0 for the passed value == 0, and 1 for the passed value &gt; 0.</td>
</tr>
<tr>
<td>sin() on page 13-160</td>
<td>Returns sine of the number.</td>
</tr>
<tr>
<td>squareroot() on page 13-160</td>
<td>Returns the square root of the number.</td>
</tr>
<tr>
<td>tan() on page 13-160</td>
<td>Returns tangent of the number.</td>
</tr>
<tr>
<td>toBytes() on page 13-160</td>
<td>Returns a Bytes object representing the Number.</td>
</tr>
<tr>
<td>toText() on page 13-160</td>
<td>Returns the number as a string formatted based on the format and NLS parameters.</td>
</tr>
<tr>
<td>trunc() on page 13-161</td>
<td>Returns a Number with the value truncated at n decimal place(s). Negative values are allowed.</td>
</tr>
</tbody>
</table>
abs()

This method returns the absolute value of the `Number` object.

**Syntax**

```cpp
const Number abs() const;
```

arcCos()

This method returns the arccosine of the `Number` object.

**Syntax**

```cpp
const Number arcCos() const;
```

arcSin()

This method returns the arcsine of the `Number` object.

**Syntax**

```cpp
const Number arcSin() const;
```

arcTan()

This method returns the arctangent of the `Number` object.

**Syntax**

```cpp
const Number arcTan() const;
```
arcTan2()

This method returns the arctangent of the Number object with the parameter specified. It returns \( \text{atan2} \ (\text{val}, x) \) where \( \text{val} \) is the parameter specified and \( x \) is the current number object.

Syntax

```cpp
const Number arcTan2(
    const Number &val) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Number parameter ( \text{val} ) to the arcTangent function ( \text{atan2}(\text{val}, x) ).</td>
</tr>
</tbody>
</table>

ceil()

This method returns the smallest integer that is greater than or equal to the Number object.

Syntax

```cpp
const Number ceil() const;
```

cos()

This method returns the cosine of the Number object.

Syntax

```cpp
const Number cos() const;
```

ext()

This method returns the natural exponential of the Number object.

Syntax

```cpp
const Number exp() const;
```

floor()

This method returns the largest integer that is less than or equal to the Number object.

Syntax

```cpp
const Number floor() const;
```

fromBytes()

This method returns a Number object represented by the byte string specified.
Syntax

```c
void fromBytes(
    const Bytes &str);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>A byte string.</td>
</tr>
</tbody>
</table>

**fromText()**

Sets `Number` object to value represented by a `string` or `UString`.

The value is interpreted based on the `fmt` and `nlsParam` parameters. In cases where `nlsParam` is not passed, the Globalization Support settings of the `envp` parameter are used.

**See Also:** *Oracle Database SQL Language Reference* for information on `TO_NUMBER`

```c
void fromText(
    const Environment *envp,
    const string &number,
    const string &fmt,
    const string &nlsParam = "");
```

```c
void fromText(
    const Environment *envp,
    const UString &number,
    const UString &fmt,
    const UString &nlsParam);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>envp</td>
<td>The OCCI environment.</td>
</tr>
<tr>
<td>number</td>
<td>The number string to be converted to a <code>Number</code> object.</td>
</tr>
<tr>
<td>fmt</td>
<td>The format string.</td>
</tr>
<tr>
<td>nlsParam</td>
<td>The NLS parameters string. If <code>nlsParam</code> is specified, this determines the NLS parameters to be used for the conversion. If <code>nlsParam</code> is not specified, the NLS parameters are picked up from <code>envp</code>.</td>
</tr>
</tbody>
</table>

**hypCos()**

This method returns the hypercosine of the `Number` object.

**Syntax**

```c
const Number hypCos() const;
```
**hypSin()**

This method returns the hypersine of the `Number` object.

**Syntax**

```cpp
const Number hypSin() const;
```

**hypTan()**

This method returns the hypertangent of the `Number` object.

**Syntax**

```cpp
const Number hypTan() const;
```

**intPower()**

This method returns a `Number` whose value is the number object raised to the power of the value specified.

**Syntax**

```cpp
const Number intPower(int val) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Power to which the number is raised.</td>
</tr>
</tbody>
</table>

**isNull()**

This method tests whether the `Number` object is NULL. If the `Number` object is NULL, then `TRUE` is returned; otherwise, `FALSE` is returned.

**Syntax**

```cpp
bool isNull() const;
```

**ln()**

This method returns the natural logarithm of the `Number` object.

**Syntax**

```cpp
const Number ln() const;
```

**log()**

This method returns the logarithm of the `Number` object with the base provided by the parameter specified.
Syntax

const Number log(
    const Number &val) const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The base to be used in the logarithm calculation.</td>
</tr>
</tbody>
</table>

operator++()

Unary operator++(). This is a postfix operator.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number &amp; operator++();</td>
<td>This method returns the Number object incremented by 1.</td>
</tr>
</tbody>
</table>
| const Number operator++(
    int incr);               | This method returns the Number object incremented by the integer specified. |

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>incr</td>
<td>The number by which the Number object is incremented.</td>
</tr>
</tbody>
</table>

operator--()

Unary operator--(). This is a prefix operator.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number &amp; operator--();</td>
<td>This method returns the Number object decremented by 1.</td>
</tr>
</tbody>
</table>
| const Number operator--(
    int decr);               | This method returns the Number object decremented by the integer specified. |

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decr</td>
<td>The number by which the Number object is decremented.</td>
</tr>
</tbody>
</table>

operator*()

This method returns the product of the parameters specified.

Syntax

Number operator*(
    const Number &first,
    const Number &second);
operator/()

This method returns the quotient of the parameters specified.

**Syntax**

```cpp
Number operator/(
    const Number &dividend,
    const Number &divisor);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dividend</td>
<td>The number to be divided.</td>
</tr>
<tr>
<td>divisor</td>
<td>The number by which to divide.</td>
</tr>
</tbody>
</table>

operator%()

This method returns the remainder of the division of the parameters specified.

**Syntax**

```cpp
Number operator%(  
    const Number &dividend,
    const Number &divisor);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dividend</td>
<td>The number to be divided.</td>
</tr>
<tr>
<td>divizor</td>
<td>The number by which to divide.</td>
</tr>
</tbody>
</table>

operator+()

This method returns the sum of the parameters specified.

**Syntax**

```cpp
Number operator+(  
    const Number &first,
    const Number &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>First number to be added.</td>
</tr>
<tr>
<td>second</td>
<td>Second number to be added.</td>
</tr>
</tbody>
</table>

operator-()

Unary `operator-()`. This method returns the negated value of the `Number` object.
Syntax

const Number operator-();

operator-()

This method returns the difference between the parameters specified.

Syntax

Number operator-(
    const Number &subtrahend,
    const Number &subtractor);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtrahend</td>
<td>The number to be reduced.</td>
</tr>
<tr>
<td>subtractor</td>
<td>The number to be subtracted.</td>
</tr>
</tbody>
</table>

operator<()

This method checks whether the first parameter specified is less than the second parameter specified. If the first parameter is less than the second parameter, then TRUE is returned; otherwise, FALSE is returned. If either parameter equals infinity, then FALSE is returned.

Syntax

bool operator<(
    const Number &first,
    const Number &second);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>First number to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>Second number to be compared.</td>
</tr>
</tbody>
</table>

operator<=()

This method checks whether the first parameter specified is less than or equal to the second parameter specified. If the first parameter is less than or equal to the second parameter, then TRUE is returned; otherwise, FALSE is returned. If either parameter equals infinity, then FALSE is returned.

Syntax

bool operator<=(
    const Number &first,
    const Number &second);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>First number to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>Second number to be compared.</td>
</tr>
</tbody>
</table>
operator>()

This method checks whether the first parameter specified is greater than the second parameter specified. If the first parameter is greater than the second parameter, then TRUE is returned; otherwise, FALSE is returned. If either parameter equals infinity, then FALSE is returned.

Syntax

```cpp
bool operator>(
    const Number &first,
    const Number &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>First number to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>Second number to be compared.</td>
</tr>
</tbody>
</table>

operator>=(())

This method checks whether the first parameter specified is greater than or equal to the second parameter specified. If the first parameter is greater than or equal to the second parameter, then TRUE is returned; otherwise, FALSE is returned. If either parameter equals infinity, then FALSE is returned.

Syntax

```cpp
bool operator>=(
    const Number &first,
    const Number &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>First number to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>Second number to be compared.</td>
</tr>
</tbody>
</table>

operator==(())

This method checks whether the parameters specified are equal. If the parameters are equal, then TRUE is returned; otherwise, FALSE is returned. If either parameter equals +infinity or -infinity, then FALSE is returned.

Syntax

```cpp
bool operator==(const Number &first, const Number &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>First number to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>Second number to be compared.</td>
</tr>
</tbody>
</table>

operator!=()  

This method checks whether the first parameter specified equals the second parameter specified. If the parameters are not equal, TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool operator!=(
    const Number &first,
    const Number &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>First number to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>Second number to be compared.</td>
</tr>
</tbody>
</table>

operator=(()  

This method assigns the value of the parameter specified to the Number object.

**Syntax**

```cpp
Number& operator=(
    const Number &num);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>A parameter of type Number.</td>
</tr>
</tbody>
</table>

operator*=(()  

This method multiplies the Number object by the parameter specified, and assigns the product to the Number object.

**Syntax**

```cpp
Number& operator*=(
    const Number &num);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>A parameter of type Number.</td>
</tr>
</tbody>
</table>

operator/=(()  

This method divides the Number object by the parameter specified, and assigns the quotient to the Number object.

**Syntax**

```cpp
Number& operator/=(
    const Number &num);
```
operator %=()

This method divides the Number object by the parameter specified, and assigns the remainder to the Number object.

**Syntax**
```
Number& operator%=(
    const Number &num);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>A parameter of type Number.</td>
</tr>
</tbody>
</table>

operator +=()

This method adds the Number object and the parameter specified, and assigns the sum to the Number object.

**Syntax**
```
Number& operator+= (
    const Number &num);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>A parameter of type Number.</td>
</tr>
</tbody>
</table>

operator -=()

This method subtracts the parameter specified from the Number object, and assigns the difference to the Number object.

**Syntax**
```
Number& operator-=(
    const Number &num);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>A parameter of type Number.</td>
</tr>
</tbody>
</table>

operator char()

This method returns the value of the Number object converted to a native char.

**Syntax**
```
operator char() const;
```
operator signed char()

This method returns the value of the Number object converted to a native signed char.

Syntax
operator signed char() const;

operator double()

This method returns the value of the Number object converted to a native double.

Syntax
operator double() const;

operator float()

This method returns the value of the Number object converted to a native float.

Syntax
operator float() const;

operator int()

This method returns the value of the Number object converted to a native int.

Syntax
operator int() const;

operator long()

This method returns the value of the Number object converted to a native long.

Syntax
operator long() const;

operator long double()

This method returns the value of the Number object converted to a native long double.

Syntax
operator long double() const;
operator short()

This method returns the value of the Number object converted to a native short integer.

Syntax
operator short() const;

operator unsigned char()

This method returns the value of the Number object converted to a native unsigned char.

Syntax
operator unsigned char() const;

operator unsigned int()

This method returns the value of the Number object converted to a native unsigned int.

Syntax
operator unsigned int() const;

operator unsigned long()

This method returns the value of the Number object converted to a native unsigned long.

Syntax
operator unsigned long() const;

operator unsigned short()

This method returns the value of the Number object converted to a native unsigned short integer.

Syntax
operator unsigned short() const;

power()

This method returns the value of the Number object raised to the power of the value provided by the parameter specified.

Syntax
const Number power(
    const Number &val) const;
### Number Class

#### prec()

This method returns the value of the `Number` object rounded to the digits of precision provided by the parameter specified.

**Syntax**

```cpp
class Number 
{
    public:
    const Number prec(
        int digits) const;
};
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>digits</td>
<td>The number of digits of precision.</td>
</tr>
</tbody>
</table>

#### round()

This method returns the value of the `Number` object rounded to the decimal place provided by the parameter specified.

**Syntax**

```cpp
class Number 
{
    public:
    const Number round(
        int decPlace) const;
};
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decPlace</td>
<td>The number of digits to the right of the decimal point.</td>
</tr>
</tbody>
</table>

#### setNull()

This method sets the value of the `Number` object to `NULL`.

**Syntax**

```cpp
class Number 
{
    public:
    void setNull();
};
```

#### shift()

This method returns the `Number` object multiplied by 10 to the power provided by the parameter specified.

**Syntax**

```cpp
class Number 
{
    public:
    const Number shift(
        int val) const;
};
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>An integer value.</td>
</tr>
</tbody>
</table>
sign()

This method returns the sign of the value of the Number object. If the Number object is negative, then create a Date object using integer parameters is returned. If the Number object equals 0, then create a Date object using integer parameters is returned. If the Number object is positive, then 1 is returned.

Syntax
const int sign() const;

sin()

This method returns the sin of the Number object.

Syntax
const Number sin() const;

squareroot()

This method returns the square root of the Number object.

Syntax
const Number squareroot() const;

tan()

This method returns the tangent of the Number object.

Syntax
const Number tan() const;

toBytes()

This method converts the Number object into a Bytes object. The bytes representation is assumed to be in length excluded format, that is, the Byte.length() method gives the length of valid bytes and the 0th byte is the exponent byte.

Syntax
Bytes toBytes() const;

toText()

Convert the Number object to a formatted string or UString based on the parameters specified.

See Also: Oracle Database SQL Language Reference for information on TO_NUMBER
This method returns the `Number` object truncated at the number of decimal places provided by the parameter specified.

### Syntax

```cpp
const Number trunc(
    int decPlace) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decPlace</td>
<td>The number of places to the right of the decimal place at which the value is to be truncated.</td>
</tr>
</tbody>
</table>
OCCI provides object navigational calls that enable applications to perform any of the following on objects:

- Creating, accessing, locking, deleting, copying, and flushing objects
- Getting references to the objects

This class enables the type definer to specify when a class can have persistent or transient instances. Instances of classes derived from `PObject` are either persistent or transient. For example, class `A` that is persistent-capable inherits from the `PObject` class:

```cpp
class A : PObject { ... }
```

The only methods valid on a `NULL` `PObject` are `setName()`, `isNull()`, and `operator=()`. Some methods, such as `lock()`, apply only for persistent instances, not for transient instances.

### Table 13–31 Enumerated Values Used by PObject Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>LockOption</td>
<td>- OCCI_LOCK_WAIT instructs the cache to pin the object only after acquiring a lock; if the object is locked by another user, the pin call with this option waits until it can acquire the lock before returning to the caller; equivalent to <code>SELECT FOR UPDATE</code></td>
</tr>
<tr>
<td></td>
<td>- OCCI_LOCK_NOWAIT instructs the cache to pin the object only after acquiring a lock; does not wait if the object is currently locked by another user; equivalent to <code>SELECT FOR UPDATE WITH NOWAIT</code></td>
</tr>
<tr>
<td>UnpinOption</td>
<td>- OCCI_PINCOUNT_RESET resets the object's pin count to 0</td>
</tr>
<tr>
<td></td>
<td>- OCCI_PINCOUNT_DECR decrements the object's pin count by 1</td>
</tr>
</tbody>
</table>

### Table 13–32 Summary of PObject Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PObject()</code> on page 13-163</td>
<td>PObject class constructor.</td>
</tr>
<tr>
<td><code>flush()</code> on page 13-163</td>
<td>Flushes a modified persistent object to the database server.</td>
</tr>
<tr>
<td><code>getConnection()</code> on page 13-163</td>
<td>Returns the connection from which the PObject object was instantiated.</td>
</tr>
<tr>
<td><code>getRef()</code> on page 13-163</td>
<td>Returns a reference to a given persistent object.</td>
</tr>
<tr>
<td><code>getSQLTypeName()</code> on page 13-164</td>
<td>Returns the Oracle database typename for this class.</td>
</tr>
<tr>
<td><code>isLocked()</code> on page 13-164</td>
<td>Tests whether the persistent object is locked.</td>
</tr>
<tr>
<td><code>isNull()</code> on page 13-164</td>
<td>Tests whether the object is <code>NULL</code>.</td>
</tr>
<tr>
<td><code>lock()</code> on page 13-164</td>
<td>Lock a persistent object on the database server. The default mode is to wait for the lock if not available.</td>
</tr>
<tr>
<td><code>markDelete()</code> on page 13-164</td>
<td>Marks a persistent object as deleted.</td>
</tr>
<tr>
<td><code>markModified()</code> on page 13-164</td>
<td>Marks a persistent object as modified or dirty.</td>
</tr>
<tr>
<td><code>operator=</code>() on page 13-165</td>
<td>Assigns one PObject to another.</td>
</tr>
</tbody>
</table>
**PObject Class**

**PObject()**

PObject class constructor.

**Syntax**

```c
PObject();
```

**Description**

Creates a NULL PObject.

**Syntax**

```c
PObject(const PObject &obj);
```

**Description**

Creates a copy of PObject.

**flush()**

This method flushes a modified persistent object to the database server.

**Syntax**

```c
void flush();
```

**getConnection()**

Returns the connection from which the persistent object was instantiated.

**Syntax**

```c
const Connection *getConnection() const;
```

**getRef()**

This method returns a reference to the persistent object.

**Syntax**

```c
RefAny getRef() const;
```
getSQLTypeName()

Returns the Oracle database typename for this class.

**Syntax**

```cpp
string getSQLTypeName() const;
```

isLocked()

This method tests whether the persistent object is locked. If the persistent object is locked, then `TRUE` is returned; otherwise, `FALSE` is returned.

**Syntax**

```cpp
bool isLocked() const;
```

isNull()

This method tests whether the persistent object is `NULL`. If the persistent object is `NULL`, then `TRUE` is returned; otherwise, `FALSE` is returned.

**Syntax**

```cpp
bool isNull() const;
```

lock()

Locks a persistent object on the database server.

**Syntax**

```cpp
void lock(
    PObject::LockOption lock_option);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock_option</td>
<td>Locking options; see Table 13-31.</td>
</tr>
</tbody>
</table>

markDelete()

This method marks a persistent object as deleted.

**Syntax**

```cpp
void markDelete();
```

markModified()

This method marks a persistent object as modified or dirty.

**Syntax**

```cpp
void mark_Modified();
```
**operator=()**

This method assigns the value of a persistent object to this `PObject` object. The nature (transient or persistent) of the object is maintained. NULL information is copied from the source instance.

**Syntax**

```cpp
PObject& operator=(
    const PObject& obj);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object from which the assigned value is obtained.</td>
</tr>
</tbody>
</table>

**operator delete()**

Deletes a persistent or transient object. The delete operator on a persistent object removes the object from the application cache only. To delete the object from the database server, invoke the `markDelete()` method.

**Syntax**

```cpp
void operator delete(
    void *obj,
    size_t size);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The pointer to object to be deleted</td>
</tr>
<tr>
<td>size</td>
<td>(Optional) Size is implicitly obtained from the object</td>
</tr>
</tbody>
</table>

**operator new()**

This method creates a new object. A persistent object is created if the connection and table name are provided. Otherwise, a transient object is created.

**Syntax**

```cpp
void *operator new(
    size_t size);
```

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Creates a default new object, with a size specification only</td>
<td></td>
</tr>
</tbody>
</table>

```cpp
void *operator new(
    size_t size,
    const Connection *conn,
    const string& tableName,
    const char *typeName);
```

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Used for creating transient objects when client side characterset is multibyte.</td>
<td></td>
</tr>
</tbody>
</table>

```cpp
void *operator new(
    size_t size,
    const Connection *conn,
    const string& tableName,
    const string& typeName,
    const string& schTableName="",
    const string& schTypeName="");
```

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Used for creating persistent objects when client side characterset is multibyte.</td>
<td></td>
</tr>
</tbody>
</table>
**pin()**

This method pins the object and increments the pin count by one. If the object is pinned, it is not freed by the cache even if there are no references to this object instance.

**Syntax**

```c
void pin();
```

**setNull()**

This method sets the object value to NULL.

**Syntax**

```c
void setNull();
```

**unmark()**

This method unmarks a persistent object as modified or deleted.

**Syntax**

```c
void unmark();
```

**unpin()**

This method unpins a persistent object. In the default mode, the pin count of the object is decremented by one. When this method is invoked with `OCCI_PINCOUNT_RESET`, the pin count of the object is reset. If the pin count is reset, this method invalidates all

---

**Syntax**

```c
void *operator new(
    size_t size,
    const Connection *conn,
    const UString& tableName,
    const UString& typeName,
    const UString& schTableName="",
    const UString& schTypeName="");
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>size of the object</td>
</tr>
<tr>
<td>conn</td>
<td>The connection to the database in which the persistent object is to be created.</td>
</tr>
<tr>
<td>tableName</td>
<td>The name of the table in the database server.</td>
</tr>
<tr>
<td>typeName</td>
<td>The SQL type name corresponding to this C++ class. The format is <code>&lt;schemaname&gt;.&lt;typename&gt;</code>.</td>
</tr>
<tr>
<td>schTableName</td>
<td>The schema table name.</td>
</tr>
<tr>
<td>schTypeName</td>
<td>The schema type name.</td>
</tr>
</tbody>
</table>
the references (Refs) pointing to this object. The cache sets the object eligible to be freed, if necessary, reclaiming memory.

**Syntax**

```c
void unpin(
    UnpinOption mode=OCCI_PINCOUNT_DECR);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>Specifies whether the UnpinOption mode, or the pin count, should be decremented or reset to 0. See Table 13–31. Valid values are OCCI_PINCOUNT_RESET and OCCI_PINCOUNT_DECR.</td>
</tr>
</tbody>
</table>
Producer Class

The Producer enqueues Messages into a queue and defines the enqueue options.

### Table 13–33 Enumerated Values Used by Producer Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
</table>
| EnqueueSequence    | ■ ENQ_BEFORE indicates that the message is enqueued before the message specified by the related message id.  
                    | ■ ENQ_TOP indicates that the message is enqueued before any other messages.  |
| Visibility         | ■ ENQ_IMMEDIATE indicates that the enqueue is not part of the current transaction. The operation constitutes a transaction of its own.  
                    | ■ ENQ_ON_COMMIT indicates that the enqueue is part of the current transaction. The operation is complete when the transaction commits. This is the default setting.  |

### Table 13–34 Summary of Producer Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer() on page 13-169</td>
<td>Producer class constructor.</td>
</tr>
<tr>
<td>getQueueName() on page 13-169</td>
<td>Retrieves the name of a queue on which the Messages is enqueued.</td>
</tr>
<tr>
<td>getRelativeMessageId() on</td>
<td>Retrieves the Message id that is referenced in a sequence deviation operation.</td>
</tr>
<tr>
<td>page 13-169</td>
<td></td>
</tr>
<tr>
<td>getSequenceDeviation() on</td>
<td>Retrieves information regarding whether the Message should be dequeued ahead of other Messages in the queue.</td>
</tr>
<tr>
<td>page 13-169</td>
<td></td>
</tr>
<tr>
<td>getTransformation() on page</td>
<td>Retrieves the transformation applied before a Message is enqueued.</td>
</tr>
<tr>
<td>13-169</td>
<td></td>
</tr>
<tr>
<td>getVisibility() on page 13-170</td>
<td>Retrieves the transactional behavior of the enqueue request.</td>
</tr>
<tr>
<td>isNull() on page 13-170</td>
<td>Tests whether the Producer is NULL.</td>
</tr>
<tr>
<td>send() on page 13-170</td>
<td>Enqueues and sends a Message.</td>
</tr>
<tr>
<td>operator=() on page 13-170</td>
<td>Assignment operator for Producer.</td>
</tr>
<tr>
<td>setNull() on page 13-171</td>
<td>Frees memory if the scope of the Producer extends beyond the Connection on which it was created.</td>
</tr>
<tr>
<td>setQueueName() on page 13-171</td>
<td>Specifies the name of a queue on which the Messages is enqueued.</td>
</tr>
<tr>
<td>setRelativeMessageId() on</td>
<td>Specifies the Message id to be referenced in the sequence deviation operation.</td>
</tr>
<tr>
<td>page 13-171</td>
<td></td>
</tr>
<tr>
<td>setSequenceDeviation() on</td>
<td>Specifies whether Message should be dequeued before other Messages in the queue.</td>
</tr>
<tr>
<td>page 13-171</td>
<td></td>
</tr>
<tr>
<td>setTransformation() on page</td>
<td>Specifies transformation applied before enqueuing a Message.</td>
</tr>
<tr>
<td>13-172</td>
<td></td>
</tr>
<tr>
<td>setVisibility() on page 13-172</td>
<td>Specifies transaction behavior of the enqueue request.</td>
</tr>
</tbody>
</table>
Producer()

Producer object constructor.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer( const Connection *conn);</td>
<td>Creates a Producer object with the specified Connection.</td>
</tr>
<tr>
<td>Producer( const Connection *conn, const string&amp; queue);</td>
<td>Creates a Producer object with the specified Connection and queue name.</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>The connection of the new Producer object.</td>
</tr>
<tr>
<td>queue</td>
<td>The queue that is used by the new Producer object.</td>
</tr>
</tbody>
</table>

getQueueName()

Retrieves the name of a queue on which the Messages are enqueued.

**Syntax**

```cpp
string getQueueName() const;
```

getRelativeMessageId()

Retrieves the Message id that is referenced in a sequence deviation operation. Used only if a sequence deviation is specified; ignored otherwise.

**Syntax**

```cpp
Bytes getRelativeMessageId() const;
```

getSequenceDeviation()

Retrieves information regarding whether the Message should be dequeued ahead of other Messages in the queue. Valid return values are `ENQ_BEFORE` and `ENQ_TOP`, as defined in Table 13-33 on page 13-168.

**Syntax**

```cpp
EnqueueSequence getSequenceDeviation() const;
```

getTransformation()

Retrieves the transformation applied before a Message is enqueued.

**Syntax**

```cpp
string getTransformation() const;
```
getVisibility()

Retrieves the transactional behavior of the enqueue request. Visibility is defined in Table 13–33 on page 13-168.

Syntax
Visibility getVisibility() const;

isNull()

Tests whether the Producer is NULL. If the Producer is NULL, then TRUE is returned; otherwise, FALSE is returned.

Syntax
bool isNull() const;

operator=(

The assignment operator for Producer.

Syntax
void operator=(
    const Producer& prod);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod</td>
<td>The original Producer</td>
</tr>
</tbody>
</table>

send()

Enqueues and sends a Message.

Syntax
Bytes send(
    Message& msg);

Bytes send(
    Message& msg,
    string& queue);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>The Message that is enqueued.</td>
</tr>
<tr>
<td>queue</td>
<td>The name of a valid queue in the database.</td>
</tr>
</tbody>
</table>
**setNull()**

Frees memory associated with the Producer. Unless working in inner scope, this call should be made before terminating the Connection.

**Syntax**

```cpp
void setNull();
```

**setQueueName()**

Specifies the name of a queue on which the Messages are enqueued. Typically used when enqueuing multiple messages to the same queue.

**Syntax**

```cpp
void setQueueName(
    const string& queue);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>The name of a valid queue in the database, to which the Messages are enqueued.</td>
</tr>
</tbody>
</table>

**setRelativeMessageId()**

Specifies the Message id to be referenced in the sequence deviation operation. If the sequence deviation is not specified, this parameter is ignored. Can be set for each enqueuing of a Message.

**Syntax**

```cpp
void setRelativeMessageId(
    const Bytes& msgid);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgid</td>
<td>The id of the relative Message.</td>
</tr>
</tbody>
</table>

**setSequenceDeviation()**

Specifies whether Message being enqueued should be dequeued before other Message(s) in the queue. Can be set for each enqueuing of a Message.

**Syntax**

```cpp
void setSequenceDeviation(
    EnqueueSequence option);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>option</td>
<td>The enqueue sequence being set, defined in Table 13–33 on page 13-168.</td>
</tr>
</tbody>
</table>
**setTransformation()**

Specifies transformation function applied before enqueuing the Message.

**Syntax**

```c
void setTransformation(
    string &fName);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fName</td>
<td>SQL transformation function.</td>
</tr>
</tbody>
</table>

**setVisibility()**

Specifies transaction behavior of the enqueue request. Can be set for each enqueuing of a Message.

**Syntax**

```c
void setVisibility(
    Visibility option);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>option</td>
<td>Visibility option being set, defined in Table 13–33 on page 13-168.</td>
</tr>
</tbody>
</table>
Ref Class

The mapping in the C++ programming language of an SQL REF value, which is a reference to an SQL structured type value in the database.

Each REF value has a unique identifier of the object it refers to. An SQL REF value may be used instead of the SQL structured type it references; it may be used as either a column value in a table or an attribute value in a structured type.

Because an SQL REF value is a logical pointer to an SQL structured type, a Ref object is by default also a logical pointer; thus, retrieving an SQL REF value as a Ref object does not materialize the attributes of the structured type on the client.

The only methods valid on a NULL Ref object are isNull(), and operator=().

A Ref object can be saved to persistent storage and is de-referenced through operator*(), operator->() or ptr() methods. T must be a class derived from PObject. In the following sections, T* and PObject* are used interchangeably.

Table 13–35 Enumerated Values Used by Ref Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>LockOptions</td>
<td>■ OCCI_LOCK_NONE clears the lock setting on the Ref object.</td>
</tr>
<tr>
<td></td>
<td>■ OCCI_LOCK_X indicates that the object should be locked, and to</td>
</tr>
<tr>
<td></td>
<td>wait for the lock to be available if the object is locked by another</td>
</tr>
<tr>
<td></td>
<td>session.</td>
</tr>
<tr>
<td></td>
<td>■ OCCI_LOCK_X_NOWAIT indicates that the object should be locked,</td>
</tr>
<tr>
<td></td>
<td>and returns an error if it is locked by another session.</td>
</tr>
<tr>
<td>PrefetchOption</td>
<td>■ OCCI_MAX_PREFETCH_DEPTH indicates that the fetch should be done to</td>
</tr>
<tr>
<td></td>
<td>maximum depth.</td>
</tr>
</tbody>
</table>

Table 13–36 Summary of Ref Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref() on page 13-174</td>
<td>Ref object constructor.</td>
</tr>
<tr>
<td>clear() on page 13-174</td>
<td>Clears the reference.</td>
</tr>
<tr>
<td>getConnection() on page 13-174</td>
<td>Returns the connection this ref was created from.</td>
</tr>
<tr>
<td>isClear() on page 13-174</td>
<td>Checks if the Ref is cleared.</td>
</tr>
<tr>
<td>isNull() on page 13-174</td>
<td>Checks if the Ref is NULL.</td>
</tr>
<tr>
<td>markDelete() on page 13-175</td>
<td>Marks the referred object as deleted.</td>
</tr>
<tr>
<td>operator-&gt;() on page 13-175</td>
<td>Dereferences the Ref and pins the object if necessary.</td>
</tr>
<tr>
<td>operator*() on page 13-175</td>
<td>Dereferences the Ref and pins or fetches the object if necessary.</td>
</tr>
<tr>
<td>operator==() on page 13-175</td>
<td>Checks if the Ref and the pointer refer to the same object.</td>
</tr>
<tr>
<td>operator!=() on page 13-176</td>
<td>Checks if the Ref and the pointer refer to different objects.</td>
</tr>
<tr>
<td>operator=() on page 13-176</td>
<td>Assignment operator.</td>
</tr>
<tr>
<td>ptr() on page 13-176</td>
<td>Returns a pointer to a PObject. Dereferences the Ref and pins or fetches the object if necessary.</td>
</tr>
<tr>
<td>setLock() on page 13-176</td>
<td>Sets the lock option for the object referred from this.</td>
</tr>
<tr>
<td>setNull() on page 13-177</td>
<td>Sets the Ref to NULL.</td>
</tr>
</tbody>
</table>
Ref()

Ref object constructor.

**Syntax**

```cpp
Ref();
Ref(const Ref<T> &src);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>The Ref that is being copied.</td>
</tr>
</tbody>
</table>

**clear()**

This method clears the Ref object.

**Syntax**

```cpp
void clear();
```

**getConnection()**

Returns the connection from which the Ref object was instantiated.

**Syntax**

```cpp
const Connection *getConnection() const;
```

**isClear()**

This method checks if Ref object is cleared.

**Syntax**

```cpp
bool isClear() const;
```

**isNull()**

This method tests whether the Ref object is NULL. If the Ref object is NULL, then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool isNull() const;
```
markDelete()

This method marks the referenced object as deleted.

Syntax

void markDelete();

operator-() 

This method dereferences the Ref object and pins, or fetches the referenced object if necessary. This might result in prefetching a graph of objects if prefetch attributes of the referenced object are set.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T *operator-&gt;();</td>
<td>Dereferences and pins or fetches a non-const Ref object.</td>
</tr>
<tr>
<td>const T *operator-&gt;() const;</td>
<td>Dereferences and pins or fetches a const Ref object.</td>
</tr>
</tbody>
</table>

operator*() 

This method dereferences the Ref object and pins or fetches the referenced object if necessary. This might result in prefetching a graph of objects if prefetch attributes of the referenced object are set. The object does not have to be deleted. Destructor would be automatically called when it goes out of scope.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&amp; operator*();</td>
<td>Dereferences and pins or fetches a non-const Ref object.</td>
</tr>
<tr>
<td>const T&amp; operator*() const;</td>
<td>Dereferences and pins or fetches a const Ref object.</td>
</tr>
</tbody>
</table>

operator==() 

This method tests whether two Ref objects are referencing the same object. If the Ref objects are referencing the same object, then TRUE is returned; otherwise, FALSE is returned.

Syntax

bool operator == ( 
   const Ref<T>& ref) const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref</td>
<td>The Ref object of the object to be compared.</td>
</tr>
</tbody>
</table>
operator!=()

This method tests whether two Ref objects are referencing the same object. If the Ref objects are not referencing the same object, then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```cpp
bool operator!= (const Ref<T> &ref) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref</td>
<td>The Ref object of the object to be compared.</td>
</tr>
</tbody>
</table>

operator=()

Assigns the Ref or the object to a Ref. For the first case, the Refs are assigned and for the second case, the Ref is constructed from the object and then assigned.

**Syntax**

```cpp
Ref<T>& operator=(const Ref<T> &src);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>The source Ref object to be assigned.</td>
</tr>
</tbody>
</table>

```cpp
Ref<T>& operator=(const T *obj);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The source object pointer whose Ref object is to be assigned.</td>
</tr>
</tbody>
</table>

ptr()

Returns a pointer to a PObject. This operator dereferences the Ref and pins or fetches the object if necessary. This might result in prefetching a graph of objects if prefetch attributes of the Ref are set.

**Syntax**

```cpp
T *ptr();
const T *ptr() const;
```

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T *ptr();</td>
<td>Returns a pointer of a non-const Ref object.</td>
</tr>
<tr>
<td>const T *ptr() const;</td>
<td>Returns a pointer of a const Ref object.</td>
</tr>
</tbody>
</table>

setLock()

This method specifies how the object should be locked when dereferenced.

**Syntax**

```cpp
void setLock(lockOptions);
```
setNull()

This method sets the Ref object to NULL.

**Syntax**

```cpp
void setNull();
```

setPrefetch()

Sets the prefetching options for complex object retrieval. This method specifies depth up to which all objects reachable from this object through Refs (transitive closure) should be prefetched. If only selected attribute types are to be prefetched, then the first version of the method must be used. This method specifies which Ref attributes of the object it refers to should be followed for prefetching of the objects (complex object retrieval) and how many levels deep those links should be followed.

**Syntax**

```cpp
void setPrefetch(
    const string &typeName,
    unsigned int depth);
```

```cpp
void setPrefetch(
    unsigned int depth);
```

```cpp
void setPrefetch(
    const string &schName,
    const string &typeName,
    unsigned int depth);
```

```cpp
void setPrefetch(
    const UString &schName,
    const UString &typeName,
    unsigned int depth);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>typeName</td>
<td>Type of the Ref attribute to be prefetched.</td>
</tr>
<tr>
<td>schName</td>
<td>Schema name of the Ref attribute to be prefetched.</td>
</tr>
<tr>
<td>depth</td>
<td>Depth level to which the links should be followed; can use PrefetchOption as defined in Table 13-35 on page 13-173.</td>
</tr>
</tbody>
</table>

unmarkDelete()

This method unmarks the referred object as dirty and available for deletion.

**Syntax**

```cpp
void unmarkDelete();
```
RefAny Class

The RefAny class is designed to support a reference to any type. Its primary purpose is to handle generic references and allow conversions of Ref in the type hierarchy. A RefAny object can be used as an intermediary between any two types, Ref<x> and Ref<y>, where x and y are different types.

Table 13–37 Summary of RefAny Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>RefAny() on page 13-178</td>
<td>Constructor for RefAny class.</td>
</tr>
<tr>
<td>clear() on page 13-178</td>
<td>Clears the reference.</td>
</tr>
<tr>
<td>getConnection() on page 13-179</td>
<td>Returns the connection this ref was created from.</td>
</tr>
<tr>
<td>isNull() on page 13-179</td>
<td>Checks if the RefAny object is NULL.</td>
</tr>
<tr>
<td>markDelete() on page 13-179</td>
<td>Marks the object as deleted.</td>
</tr>
<tr>
<td>operator=() on page 13-179</td>
<td>Assignment operator for RefAny.</td>
</tr>
<tr>
<td>operator==() on page 13-179</td>
<td>Checks if this RefAny object equals a specified RefAny.</td>
</tr>
<tr>
<td>operator!=() on page 13-180</td>
<td>Checks if not equal.</td>
</tr>
<tr>
<td>unmarkDelete() on page 13-180</td>
<td>Unmarks the object as deleted.</td>
</tr>
</tbody>
</table>

RefAny()

A Ref<T> can always be converted to a RefAny; there is a method to perform the conversion in the Ref<T> template. Each Ref<T> has a constructor and assignment operator that takes a reference to RefAny.

Syntax Description

```
RefAny();

RefAny(
    const Connection *sessptr,
    const OCIRef *ref);

RefAny(
    const RefAny& src);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sessptr</td>
<td>Session pointer</td>
</tr>
<tr>
<td>ref</td>
<td>A reference</td>
</tr>
<tr>
<td>src</td>
<td>The source RefAny object to be assigned</td>
</tr>
</tbody>
</table>

clear()

This method clears the reference.
**Syntax**

```cpp
void clear();
```

**getConnection()**

Returns the connection from which this reference was instantiated.

**Syntax**

```cpp
const Connection* getConnection() const;
```

**isNull()**

Returns TRUE if the object pointed to by this ref is NULL else FALSE.

**Syntax**

```cpp
bool isNull() const;
```

**markDelete()**

This method marks the referred object as deleted.

**Syntax**

```cpp
void markDelete();
```

**operator=()**

Assignment operator for RefAny.

**Syntax**

```cpp
RefAny& operator=(
    const RefAny& src);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>The source RefAny object to be assigned.</td>
</tr>
</tbody>
</table>

**operator==( )**

Compares this ref with a RefAny object and returns TRUE if both the refs are referring to the same object in the cache, otherwise it returns FALSE.

**Syntax**

```cpp
bool operator== (    
    const RefAny &refAnyR) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refAnyR</td>
<td>RefAny object to which the comparison is made.</td>
</tr>
</tbody>
</table>
operator!=( )

Compares this ref with the RefAny object and returns TRUE if both the refs are not referring to the same object in the cache, otherwise it returns FALSE.

**Syntax**

```cpp
bool operator!=( const RefAny &refAnyR ) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refAnyR</td>
<td>RefAny object to which the comparison is made.</td>
</tr>
</tbody>
</table>

unmarkDelete( )

This method unmarks the referred object as dirty.

**Syntax**

```cpp
void unmarkDelete();
```
ResultSet Class

A ResultSet provides access to a table of data generated by executing a Statement. Table rows are retrieved in sequence. Within a row, column values can be accessed in any order.

A ResultSet maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The `next` method moves the cursor to the next row.

The `getxxx()` methods retrieve column values for the current row. You can retrieve values using the index number of the column. Columns are numbered beginning at 1. For the `getxxx()` methods, OCII attempts to convert the underlying data to the specified C++ type and returns a C++ value. SQL types are mapped to C++ types with the `ResultSet::getxxx()` methods.

The number, types and properties of a ResultSet's columns are provided by the `MetaData` object returned by the `getColumnListMetaData()` method.

**Table 13-38 Enumerated Values Used by ResultSet Class**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
</table>
| Status    | ■ DATA_AVAILABLE indicates that data for one or more rows was successfully fetched from the server; up to the requested number of rows (`numRows`) were returned. When in streamed mode, use the `getNumArrayRows()` method to determine the exact number of rows retrieved when `numRows` is greater than 1.  
 ■ STREAM_DATA_AVAILABLE indicates that the application should call the `getCurrentStreamColumn()` method and read stream.  
 ■ END_OF_FETCH indicates that no data was available for fetching. |

**Table 13-39 Summary of ResultSet Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cancel()</code> on page 13-183</td>
<td>Cancels the ResultSet.</td>
</tr>
<tr>
<td><code>closeStream()</code> on page 13-183</td>
<td>Closes the specified Stream.</td>
</tr>
<tr>
<td><code>getBDouble()</code> on page 13-184</td>
<td>Returns the value of a column in the current row as a BDouble.</td>
</tr>
<tr>
<td><code>getBfile()</code> on page 13-184</td>
<td>Returns the value of a column in the current row as a Bfile.</td>
</tr>
<tr>
<td><code>getBFloat()</code> on page 13-184</td>
<td>Returns the value of a column in the current row as a BFloat.</td>
</tr>
<tr>
<td><code>getBlob()</code> on page 13-185</td>
<td>Returns the value of a column in the current row as a Blob object.</td>
</tr>
<tr>
<td><code>getBytes()</code> on page 13-185</td>
<td>Returns the value of a column in the current row as a Bytes array.</td>
</tr>
<tr>
<td><code>getCharSet()</code> on page 13-185</td>
<td>Returns the character set in which data would be fetched.</td>
</tr>
<tr>
<td><code>getCharSetUString()</code> on page 13-185</td>
<td>Returns the character set in which data would be fetched as a UString.</td>
</tr>
<tr>
<td><code>getClob()</code> on page 13-186</td>
<td>Returns the value of a column in the current row as a Clob object.</td>
</tr>
</tbody>
</table>
### Table 13–39  (Cont.) Summary of ResultSet Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getColumnListMetaData()</code> on page 13-186</td>
<td>Returns the describe information of the result set columns as a <code>MetaData</code> object.</td>
</tr>
<tr>
<td><code>getCurrentStreamColumn()</code> on page 13-186</td>
<td>Returns the column index of the current readable Stream.</td>
</tr>
<tr>
<td><code>getCurrentStreamRow()</code> on page 13-186</td>
<td>Returns the current row of the <code>ResultSet</code> being processed.</td>
</tr>
<tr>
<td><code>getCursor()</code> on page 13-186</td>
<td>Returns the nested cursor as a <code>ResultSet</code>.</td>
</tr>
<tr>
<td><code>getDate()</code> on page 13-187</td>
<td>Returns the value of a column in the current row as a <code>Date</code> object.</td>
</tr>
<tr>
<td><code>getDatabaseNCHARParam()</code> on page 13-187</td>
<td>Returns whether data is in NCHAR character set or not.</td>
</tr>
<tr>
<td><code>getDouble()</code> on page 13-187</td>
<td>Returns the value of a column in the current row as a C++ double.</td>
</tr>
<tr>
<td><code>getFloat()</code> on page 13-188</td>
<td>Returns the value of a column in the current row as a C++ float.</td>
</tr>
<tr>
<td><code>getInt()</code> on page 13-188</td>
<td>Returns the value of a column in the current row as a C++ int.</td>
</tr>
<tr>
<td><code>getIntervalDS()</code> on page 13-188</td>
<td>Returns the value of a column in the current row as a <code>IntervalDS</code>.</td>
</tr>
<tr>
<td><code>getIntervalYM()</code> on page 13-188</td>
<td>Returns the value of a column in the current row as a <code>IntervalYM</code>.</td>
</tr>
<tr>
<td><code>getMaxColumnSize()</code> on page 13-189</td>
<td>Returns the maximum amount of data to read from a column.</td>
</tr>
<tr>
<td><code>getNumArrayRows()</code> on page 13-189</td>
<td>Returns the actual number of rows fetched in the last array fetch.</td>
</tr>
<tr>
<td><code>getNumber()</code> on page 13-189</td>
<td>Returns the value of a column in the current row as a <code>Number</code> object.</td>
</tr>
<tr>
<td><code>getObject()</code> on page 13-189</td>
<td>Returns the value of a column in the current row as a <code>PObject</code>.</td>
</tr>
<tr>
<td><code>getRef()</code> on page 13-190</td>
<td>Returns the value of a column in the current row as a <code>Ref</code>.</td>
</tr>
<tr>
<td><code>getRowid()</code> on page 13-190</td>
<td>Returns the current <code>ROWID</code> for a <code>SELECT FOR UPDATE</code> statement.</td>
</tr>
<tr>
<td><code>getRowPosition()</code> on page 13-190</td>
<td>Returns the row id of the current row position.</td>
</tr>
<tr>
<td><code>getStatement()</code> on page 13-190</td>
<td>Returns the <code>Statement</code> of the <code>ResultSet</code>.</td>
</tr>
<tr>
<td><code>getStream()</code> on page 13-191</td>
<td>Returns the value of a column in the current row as a <code>Stream</code>.</td>
</tr>
<tr>
<td><code>getString()</code> on page 13-191</td>
<td>Returns the value of a column in the current row as a string.</td>
</tr>
<tr>
<td><code>getTimestamp()</code> on page 13-191</td>
<td>Returns the value of a column in the current row as a <code>Timestamp</code> object.</td>
</tr>
<tr>
<td><code>getUInt()</code> on page 13-191</td>
<td>Returns the value of a column in the current row as a C++ unsigned int</td>
</tr>
<tr>
<td><code>getUString()</code> on page 13-192</td>
<td>Returns the value of a column in the current row as a <code>UString</code>.</td>
</tr>
<tr>
<td><code>getVector()</code> on page 13-192</td>
<td>Returns the specified collection parameter as a vector.</td>
</tr>
</tbody>
</table>
**cancel()**

This method cancels the result set.

**Syntax**

```java
void cancel();
```

**closeStream()**

This method closes the stream specified by the parameter `stream`.

---

### Table 13–39  (Cont.) Summary of ResultSet Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getVectorOfRefs()</code> on page 13-194</td>
<td>Returns the column in the current position as a vector of <code>Refs</code>.</td>
</tr>
<tr>
<td><code>isNull()</code> on page 13-194</td>
<td>Checks whether the value is <code>NULL</code>.</td>
</tr>
<tr>
<td><code>isTruncated()</code> on page 13-195</td>
<td>Checks whether truncation has occurred.</td>
</tr>
<tr>
<td><code>next()</code> on page 13-195</td>
<td>Makes the next row the current row in a <code>ResultSet</code>.</td>
</tr>
<tr>
<td><code>preTruncationLength()</code> on page 13-195</td>
<td>Returns the actual length of the parameter before truncation.</td>
</tr>
<tr>
<td><code>setBinaryStreamMode()</code> on page 13-196</td>
<td>Specifies that a column is to be returned as a binary stream.</td>
</tr>
<tr>
<td><code>setCharacterStreamMode()</code> on page 13-196</td>
<td>Specifies that a column is to be returned as a character stream.</td>
</tr>
<tr>
<td><code>setCharSet()</code> on page 13-196</td>
<td>Specifies the character set in which the data is to be returned.</td>
</tr>
<tr>
<td><code>setCharSetUString()</code> on page 13-196</td>
<td>Specifies the character set in which the data is to be returned.</td>
</tr>
<tr>
<td><code>setDatabaseNCHARParam()</code> on page 13-197</td>
<td>If the parameter is going to be retrieved from a column that contains data in the database's NCHAR character set, then OCCI must be informed by passing a true value.</td>
</tr>
<tr>
<td><code>setDataBuffer()</code> on page 13-197</td>
<td>Specifies the data buffer into which data is to be fetched, or the gather and scatter binds and defines made.</td>
</tr>
<tr>
<td><code>setErrorOnNull()</code> on page 13-198</td>
<td>Enables Or Disables exception when <code>NULL</code> value is read.</td>
</tr>
<tr>
<td><code>setErrorOnTruncate()</code> on page 13-198</td>
<td>Enables Or Disables exception when truncation occurs.</td>
</tr>
<tr>
<td><code>setPrefetchMemorySize()</code> on page 13-198</td>
<td>Sets the amount of memory that is used internally by OCCI to store data fetched during each round trip to the server.</td>
</tr>
<tr>
<td><code>setPrefetchRowCount()</code> on page 13-199</td>
<td>Sets the number of rows that are fetched internally by OCCI during each round trip to the server.</td>
</tr>
<tr>
<td><code>setMaxColumnSize()</code> on page 13-199</td>
<td>Specifies the maximum amount of data to read from a column.</td>
</tr>
<tr>
<td><code>status()</code> on page 13-199</td>
<td>Returns the current status of the <code>ResultSet</code>.</td>
</tr>
</tbody>
</table>
getBDouble()

Syntax
void closeStream(
    Stream *stream);

Parameter | Description
---|---
stream | The Stream to be closed.

getBDouble()

This method returns the value of a column in the current row as a BDouble. If the value is SQL NULL, the result is NULL.

Syntax
BDouble getBDouble(
    unsigned int colIndex);

Parameter | Description
---|---
colIndex | Column index, first column is 1, second is 2, and so on.

getBFile()

This method returns the value of a column in the current row as a Bfile. Returns the column value; if the value is SQL NULL, the result is NULL.

Syntax
Bfile getBfile(
    unsigned int colIndex);

Parameter | Description
---|---
colIndex | Column index, first column is 1, second is 2, and so on.

getBFloat()

This method returns the value of a column in the current row as a BFloat. If the value is SQL NULL, the result is NULL.

Syntax
BFloat getBFloat(
    unsigned int colIndex);

Parameter | Description
---|---
colIndex | Column index, first column is 1, second is 2, and so on.
**getBlob()**

Get the value of a column in the current row as a Blob. Returns the column value; if the value is SQL NULL, the result is NULL.

**Syntax**

```cpp
Blob getBlob(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getBytes()**

Get the value of a column in the current row as a Bytes array. The bytes represent the raw values returned by the server. Returns the column value; if the value is SQL NULL, the result is NULL array

**Syntax**

```cpp
Bytes getBytes(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getCharSet()**

Gets the character set in which data would be fetched, as a string.

**Syntax**

```cpp
string getCharSet(
    unsigned int colIndex) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getCharSetUString()**

Gets the character set in which data would be fetched, as a string.

**Syntax**

```cpp
UString getCharSetUString(
    unsigned int colIndex) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>
getClob()

Get the value of a column in the current row as a Clob. Returns the column value; if the value is SQL NULL, the result is NULL.

Syntax

```cpp
Clob getClob(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

gNullableListMetaData()

The number, types and properties of a ResultSet's columns are provided by the getMetaData method. Returns the description of a ResultSet's columns. This method returns the value of the given column as a PObject. The type of the C++ object is the C++ PObject type corresponding to the column's SQL type registered with Environment's map. This method is used to materialize data of SQL user-defined types.

Syntax

```cpp
vector<MetaData> getColumnListMetaData() const;
```

gCurrentStreamColumn()

If the result set has any input Stream parameters, this method returns the column index of the current input Stream that must be read. If no output Stream must be read, or there are no input Stream columns in the result set, this method returns 0. Returns the column index of the current input Stream column that must be read.

Syntax

```cpp
unsigned int getCurrentStreamColumn() const;
```

gCurrentStreamRow()

If the result has any input Streams, this method returns the current row of the result set that is being processed by OCCI. If this method is called after all the rows in the set of array of rows have been processed, it returns 0. Returns the row number of the current row that is being processed. The first row is numbered 1 and so on.

Syntax

```cpp
unsigned int getCurrentStreamRow() const;
```

gCursor()

Get the nested cursor as a ResultSet. Data can be fetched from this result set. A nested cursor results from a nested query with a CURSOR(SELECT...) clause:
SELECT last_name,
       CURSOR(SELECT  department_name FROM departments)
FROM employees WHERE last_name = 'JONES'

Note that if there are multiple REF CURSORs being returned, data from each cursor must be completely fetched before retrieving the next REF CURSOR and starting fetch on it. Returns A ResultSet for the nested cursor.

**Syntax**

```
ResultSet * getCursor(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getDatabaseNCHARParam()**

Returns whether data is in NCHAR character set or not.

**Syntax**

```
bool getDatabaseNCHARParam(
    unsigned int paramIndex) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index, first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getDouble()**

Gets the value of a column in the current row as a C++ double. Returns the column value; if the value is SQL NULL, the result is 0.

**Syntax**

```
double getDouble(
    unsigned int colIndex);
```
getFloat()

Get the value of a column in the current row as a C++ float. Returns the column value; if the value is SQL NULL, the result is 0.

**Syntax**

```cpp
float getFloat(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getInt()

Get the value of a column in the current row as a C++ int. Returns the column value; if the value is SQL NULL, the result is 0.

**Syntax**

```cpp
int getInt(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getIntervalDS()

Get the value of a column in the current row as a IntervalDS object. Returns the column value; if the value is SQL NULL, the result is NULL.

**Syntax**

```cpp
IntervalDS getIntervalDS(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getIntervalYM()

Get the value of a column in the current row as a IntervalYM object. Returns the column value; if the value is SQL NULL, the result is NULL.
### ResultSet Class

#### IntervalYM

**Syntax**

```c
IntervalYM getIntervalYM(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

#### getMaxColumnSize()

Get the maximum amount of data to read for a column.

**Syntax**

```c
unsigned int getMaxColumnSize(
    unsigned int colIndex) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

#### getNumArrayRows()

Returns the actual number of rows fetched in the last array fetch. Used in conjunction with the `next()` method. This method cannot be used for non-array fetches.

**Syntax**

```c
unsigned int getNumArrayRows() const;
```

#### getNumber()

Get the value of a column in the current row as a `Number` object. Returns the column value; if the value is SQL `NULL`, the result is `NULL`.

**Syntax**

```c
Number getNumber(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

#### getObject()

Returns a pointer to a `PObject` holding the column value.

**Syntax**

```c
PObject * getObject(
    unsigned int colIndex);
```
getRef()

Get the value of a column in the current row as a RefAny. Retrieving a Ref value does not materialize the data to which Ref refers. Also the Ref value remains valid while the session or connection on which it is created is open. Returns a RefAny holding the column value.

**Syntax**

```
RefAny getRef(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index; first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getRowid()

Get the current row id for a SELECT...FOR UPDATE statement. The row id can be bound to a prepared DELETE statement and so on. Returns current rowid for a SELECT...FOR UPDATE statement.

**Syntax**

```
Bytes getRowid(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getRowPosition()

Get the rowid of the current row position.

**Syntax**

```
Bytes getRowPosition() const;
```

getAddress()

This method returns the statement of the ResultSet.

**Syntax**

```
Statement* getStatement() const;
```
**getStream()**

This method returns the value of a column in the current row as a `Stream`.

**Syntax**

```c
Stream * getStream(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getString()**

Get the value of a column in the current row as a string. Returns the column value; if the value is SQL `NULL`, the result is an empty string.

**Syntax**

```c
string getString(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getTimestamp()**

Get the value of a column in the current row as a `Timestamp` object. Returns the column value; if the value is SQL `NULL`, the result is `NULL`.

**Syntax**

```c
Timestamp getTimestamp(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getUInt()**

Get the value of a column in the current row as a C++ `int`. Returns the column value; if the value is SQL `NULL`, the result is 0.

**Syntax**

```c
unsigned int getUInt(
    unsigned int colIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>
getUString()

Returns the value as a UString.
This method should be called only if the environment’s character set is UTF16, or if setCharset() method has been called to explicitly retrieve UTF16 data.

Syntax
UString getUString(
    unsigned int colIndex);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index; first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getVector()

This method returns the column in the current position as a vector. The column should be a collection type (varray or nested table). The SQL type of the elements in the collection should be compatible with the data type of the objects in the vector.

Syntax Description

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| void getVector(
    ResultSet *rs,
    unsigned int colIndex,
    vector<BDouble> &vect); | Used for BDouble vectors. |
| void getVector(
    ResultSet *rs,
    unsigned int colIndex,
    vector<Bfile> &vect); | Used for Bfile vectors. |
| void getVector(
    ResultSet *rs,
    unsigned int colIndex,
    vector<BFloat> &vect); | Used for BFloat vectors. |
| void getVector(
    ResultSet *rs,
    unsigned int colIndex,
    vector<Blob> &vect); | Used for Blob vectors. |
| void getVector(
    ResultSet *rs,
    unsigned int colIndex,
    vector<Clob> &vect); | Used for Clob vectors. |
| void getVector(
    ResultSet *rs,
    unsigned int colIndex,
    vector<Date> &vect); | Used for vectors of Date Class. |
| void getVector(
    ResultSet *rs,
    unsigned int colIndex,
    vector<double> &vect); | Used for vectors of double type. |
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`void getVector(</td>
<td>Used for vectors of float type.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;float&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Used for vectors of int type.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;int&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Used for vectors of IntervalDS Class.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;IntervalDS&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Used for vectors of IntervalYM Class.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;IntervalYM&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Used for vectors of Number Class.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;Number&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Available only on platforms where partial ordering of function templates is supported. This function may be deprecated in the future. getVectorOfRefs() can be used instead.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;Ref&lt;T&gt;&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Used for vectors of RefAny Class.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;RefAny&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Used for vectors of string type.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;string&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Intended for use on platforms where partial ordering of function templates is supported.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;T *&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Intended for use on platforms where partial ordering of function templates is not supported, such as Windows NT and z/OS.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;T&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Used for vectors of Timestamp Class.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;Timestamp&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>`void getVector(</td>
<td>Used for vectors of unsigned int type.</td>
</tr>
<tr>
<td>ResultSet *rs,</td>
<td></td>
</tr>
<tr>
<td>unsigned int colIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;unsigned int&gt; &amp;vect);</td>
<td></td>
</tr>
</tbody>
</table>
getVectorOfRefs()

Returns the column in the current position as a vector of REFS. The column should be a collection type (varray or nested table) of REFS. It is recommend to use this function instead of specialized method getVector() for Ref<T>.

Syntax

```cpp
void getVectorOfRefs(
    ResultSet *rs,
    unsigned int colIndex,
    vector< Ref<T> > &vect);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>The result set</td>
</tr>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>vect</td>
<td>The reference to the vector (OUT parameter).</td>
</tr>
</tbody>
</table>

isNull()

A column may have the value of SQL NULL; isNull() reports whether the last column read had this special value. Note that you must first call get.xxx() on a column to try to read its value and then call isNull() to find if the value was the SQL NULL. Returns TRUE if last column read was SQL NULL.

Syntax

```cpp
bool isNull(
    unsigned int colIndex) const;
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>
**isTruncated()**

This method checks whether the value of the parameter is truncated. If the value of the parameter is truncated, then TRUE is returned; otherwise, FALSE is returned.

**Syntax**

```c++
bool isTruncated(
    unsigned int paramIndex) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index, first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**next()**

This method fetches a specified number of rows, `numRows`, from a previously executed query, and reports the status of this fetch as defined in Table 13–38.

For non-streamed mode, `next()` only returns the status of `DATA_AVAILABLE` or `END_OF_FETCH`.

- When fetching one row at a time (`numRows=1`), process the data using `getxxx()` methods.
- When fetching several rows at once (`numRows>1`), as in an Array Fetch, you must use the `setDataBuffer()` method to specify the location of your preallocated buffers before invoking `next()`.

Up to `numRows` data records would populate the buffers specified by the `setDataBuffer()` call. To determine exactly how many records were returned, use the `getNumArrayRows()` method.

**Syntax**

```c++
Status next(
    unsigned int numRows = 1);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numRows</td>
<td>Number of rows to fetch for array fetches.</td>
</tr>
</tbody>
</table>

**preTruncationLength()**

Returns the actual length of the parameter before truncation.

**Syntax**

```c++
int preTruncationLength(
    unsigned int paramIndex) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index, first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>
**setBinaryStreamMode()**

Defines that a column is to be returned as a binary stream by the `getStream` method.

**Syntax**

```cpp
void setBinaryStreamMode(
    unsigned int colIndex,
    unsigned int size);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>size</td>
<td>The amount of data to be read as a binary stream.</td>
</tr>
</tbody>
</table>

**setCharacterStreamMode()**

Defines that a column is to be returned as a character stream by the `getStream()` method.

**Syntax**

```cpp
void setCharacterStreamMode(
    unsigned int colIndex,
    unsigned int size);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>size</td>
<td>The amount of data to be read as a character stream.</td>
</tr>
</tbody>
</table>

**setCharSet()**

Overrides the default character set for the specified column. Data is converted from the database character set to the specified character set for this column.

**Syntax**

```cpp
void setCharSet(
    unsigned int colIndex,
    string charSet);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>charSet</td>
<td>Desired character set, as a string.</td>
</tr>
</tbody>
</table>

**setCharSetUString()**

Specifies the character set value as a `UString` in which the data is returned.
**ResultSet Class**

**OCCI Application Programming Interface**

**Syntax**

```cpp
UString setCharSetUString(
    unsigned int colIndex,
    const UString &charSet);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>charSet</td>
<td>Desired character set, as a string.</td>
</tr>
</tbody>
</table>

**setDatabaseNCHARParam()**

If the parameter is going to be retrieved from a column that contains data in the database's NCHAR character set, then OCCI must be informed by passing a TRUE value. A FALSE can be passed to restore the default.

**Syntax**

```cpp
void setDatabaseNCHARParam(
    unsigned int paramIndex,
    bool isNCHAR);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index, first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>isNCHAR</td>
<td>TRUE or FALSE.</td>
</tr>
</tbody>
</table>

**setDataBuffer()**

Specifies a data buffer where data would be fetched or bound. The `buffer` parameter is a pointer to a user allocated data buffer. The current length of data must be specified in the `length` parameter. The amount of data should not exceed the `size` parameter. Finally, `type` is the data type of the data. Only non OCCI and non C++ specific types can be used, such as STL string. OCCI classes like `Bytes` and `Date` cannot be used.

If `setDataBuffer()` is used to fetch data for array fetches, it should be called only once for each result set. Data for each row is assumed to be at `buffer(i-1)` location, where `i` is the row number. Similarly, the length of the data would be assumed to be at `(length+(i-1))`.

**Syntax**

```cpp
void setDataBuffer(
    unsigned int colIndex,
    void *buffer,
    Type type,
    sb4 size = 0,
    ub2 *length = NULL,
    sb2 *ind = NULL,
    ub2 *rc = NULL);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>
setErrorOnNull()

This method enables/disables exceptions for reading of NULL values on colIndex column of the result set.

Syntax

void setErrorOnNull(
    unsigned int colIndex,
    bool causeException);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>causeException</td>
<td>Enable exceptions if TRUE. Disable if FALSE.</td>
</tr>
</tbody>
</table>

setErrorOnTruncate()

This method enables/disables exceptions when truncation occurs.

Syntax

void setErrorOnTruncate(
    unsigned int paramIndex,
    bool causeException);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index, first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>causeException</td>
<td>Enable exceptions if TRUE. Disable if FALSE.</td>
</tr>
</tbody>
</table>

setPrefetchMemorySize()

Sets the amount of memory that is used internally by OCCI to store data fetched during each round trip to the server. A value of 0 means that the amount of data

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>Pointer to user-allocated buffer. For array fetches, it should have numRows * size bytes in it. For gather or scatter binds and defines, this structure stores the address of OCIIOVec and the number of OCIIOVec elements that start at that address.</td>
</tr>
<tr>
<td>type</td>
<td>Type of the data that is provided (or retrieved) in the buffer.</td>
</tr>
<tr>
<td>size</td>
<td>Size of the data buffer. For array fetches, it is the size of each element of the data items.</td>
</tr>
<tr>
<td>length</td>
<td>Pointer to the length of data in the buffer; for array fetches, it should be an array of length data for each buffer element; the size of the array should be equal to arrayLength.</td>
</tr>
<tr>
<td>ind</td>
<td>Pointer to an indicator variable or array (IN/OUT).</td>
</tr>
<tr>
<td>rc</td>
<td>Pointer to array of column level return codes (OUT).</td>
</tr>
</tbody>
</table>
fetched during the round trip is constrained by the `FetchRowCount` parameter. If both parameters are nonzero, the smaller of the two is used.

**Syntax**

```c
void setPrefetchMemorySize(
    unsigned int bytes);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>Number of bytes used for storing data fetched during each server round trip.</td>
</tr>
</tbody>
</table>

**setPrefetchRowCount()**

Sets the number of rows that are fetched internally by OCCI during each round trip to the server. A value of 0 means that the amount of data fetched during the round trip is constrained by the `FetchMemorySize` parameter. If both parameters are nonzero, the smaller of the two is used. If both of these parameters are zero, row count internally defaults to 1 row and that is the value returned from the `getFetchRowCount()` method.

**Syntax**

```c
void setPrefetchRowCount(
    unsigned int rowCount);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowCount</td>
<td>Number of rows to fetch for each round trip to the server.</td>
</tr>
</tbody>
</table>

**setMaxColumnSize()**

Sets the maximum amount of data to read for a column.

**Syntax**

```c
void setMaxColumnSize(
    unsigned int colIndex,
    unsigned int max);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index, first column is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>max</td>
<td>The maximum amount of data to be read.</td>
</tr>
</tbody>
</table>

**status()**

Returns the current `Status` of the result set, as defined in Table 13–38. This method can be called repeatedly.

**Syntax**

```c
Status status() const;
```
SQLException Class

The SQLException class provides information on generated errors, their codes and associated messages.

Table 13–40  Summary of SQLException

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLException() on page 13-200</td>
<td>SQLException constructor.</td>
</tr>
<tr>
<td>getErrorCode() on page 13-200</td>
<td>Returns the database error code.</td>
</tr>
<tr>
<td>getMessage() on page 13-201</td>
<td>Returns the error message string for this exception.</td>
</tr>
<tr>
<td>getNLSMessage() on page 13-201</td>
<td>Returns the error message string for this exception (Unicode support).</td>
</tr>
<tr>
<td>getNLSUStringMessage() on page 13-201</td>
<td>Returns the error message UString for this exception (Unicode support).</td>
</tr>
<tr>
<td>getUStringMessage() on page 13-201</td>
<td>Returns the error message UString for this exception.</td>
</tr>
<tr>
<td>getXAErrorCode() on page 13-201</td>
<td>Returns the error message string for this exception.</td>
</tr>
<tr>
<td>setErrorCtx() on page 13-202</td>
<td>Sets the error context.</td>
</tr>
<tr>
<td>what() on page 13-202</td>
<td>Returns the error message associated with the SQLException.</td>
</tr>
</tbody>
</table>

SQLException()

This is the SQLException constructor.

Syntax

<table>
<thead>
<tr>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructs a NULL SQLException object.</td>
<td>SQLException();</td>
</tr>
<tr>
<td>Constructs an SQLException object as a copy of another SQLException object.</td>
<td>SQLException( const SQLException &amp;e);</td>
</tr>
</tbody>
</table>

Parameter Description

e  The SQLException to be copied.

getErrorCode()

Gets the database error code.

Syntax

```cpp
cint getErrorCode() const;
```
getMessage()

Returns the error message string of this SQLException if it was created with an error message string. Returns NULL if the SQLException was created with no error message.

**Syntax**

```c
string getMessage() const;
```

getNLSMessage()

Returns the error message string of this SQLException if it was created with an error message string. Passes the globalization enabled environment. Returns a NULL string if the SQLException was created with no error message. The error message is in the character set associated with the environment.

**Syntax**

```c
string getNLSMessage(
    Environment *env) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>The globalization enabled environment.</td>
</tr>
</tbody>
</table>

getNLSUStringMessage()

Returns the error message UString of this SQLException if it was created with an error message UString. Passes the globalization enabled environment. Returns a NULL UString if the SQLException was created with no error message. The error message is in the character set associated with the environment.

**Syntax**

```c
UString getNLSUStringMessage(
    Environment *env) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>The globalization enabled environment.</td>
</tr>
</tbody>
</table>

getUStringMessage()

Returns the error message UString of this SQLException if it was created with an error message UString. Returns a NULL UString if the SQLException was created with no error message. The error message is in the character set associated with the environment.

**Syntax**

```c
UString getUStringMessage() const;
```
getXAErrorCode()

Determine if the thrown exception is due to an XA or an SQL error.
Used by C++ XA applications with dynamic registration. Returns an XA error code if
the exception is due to XA, or XA_OK otherwise.

**Syntax**

```cpp
int getXAErrorCode(
    const string &dbname) const;
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbname</td>
<td>The database name; same as the optional dbname provided in the Open String and used when connecting to the Resource Manager.</td>
</tr>
</tbody>
</table>

setErrorCtx()

Sets the pointer to the error context.

**Syntax**

```cpp
void setErrorCtx(
    void *ctx);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>The pointer to the error context.</td>
</tr>
</tbody>
</table>

what()

Standard C++ compliant function; returns the error message associated with the SQLException.

**Syntax**

```cpp
const char *what() const throw();
```
StatelessConnectionPool Class

This class represents a pool of stateless, authenticated connections to the database.

Table 13–41  Enumerated Values Used by StatelessConnectionPool Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>PoolType</td>
<td>■ HETEROGENEOUS is the default state; connections with different</td>
</tr>
<tr>
<td></td>
<td>authentication contexts can be created in the same pool. This pool</td>
</tr>
<tr>
<td></td>
<td>type also supports external authentication.</td>
</tr>
<tr>
<td></td>
<td>■ HOMOGENEOUS indicates that all connections in the pool are</td>
</tr>
<tr>
<td></td>
<td>authenticated with the username and password provided during</td>
</tr>
<tr>
<td></td>
<td>pool creation. No proxy connections can be created. minConn and</td>
</tr>
<tr>
<td></td>
<td>incrConn values are considered only in these HOMOGENEOUS pools.</td>
</tr>
<tr>
<td></td>
<td>■ NO_RLB turns off run-time load balancing in the connection pool.</td>
</tr>
<tr>
<td></td>
<td>Can be used with both HETEROGENEOUS and HOMOGENEOUS pools.</td>
</tr>
<tr>
<td></td>
<td>■ USES_EXT_AUTH indicates that the connections in the pool support</td>
</tr>
<tr>
<td></td>
<td>external authentication. Can only be used with HETEROGENEOUS pools.</td>
</tr>
<tr>
<td>BusyOption</td>
<td>■ WAIT indicates that the thread waits and blocks until the connection</td>
</tr>
<tr>
<td></td>
<td>becomes free.</td>
</tr>
<tr>
<td></td>
<td>■ NOWAIT throws and error.</td>
</tr>
<tr>
<td></td>
<td>■ FORCEGET indicates that a new connection is created, even when</td>
</tr>
<tr>
<td></td>
<td>maximum number of connections is opened and all are busy.</td>
</tr>
<tr>
<td>DestroyMode</td>
<td>■ DEFAULT indicates that if any active busy connections in the</td>
</tr>
<tr>
<td></td>
<td>pool, ORA24422 error is thrown.</td>
</tr>
<tr>
<td></td>
<td>■ SPD_FORCE indicates that any busy connections in the pool are</td>
</tr>
<tr>
<td></td>
<td>forcefully terminated and the pool is destroyed; the user loses</td>
</tr>
<tr>
<td></td>
<td>memory corresponding to the number of connections forcefully</td>
</tr>
<tr>
<td></td>
<td>terminated.</td>
</tr>
</tbody>
</table>

Table 13–42  Summary of StatelessConnectionPool Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAnyTaggedConnection() on page 13-204</td>
<td>Returns a pointer to the connection object, without the restriction of a matching tag.</td>
</tr>
<tr>
<td>getAnyTaggedProxyConnection() on page 13-205</td>
<td>Returns a proxy connection from a connection pool.</td>
</tr>
<tr>
<td>getBusyConnections() on page 13-206</td>
<td>Returns the number of busy connections in the connection pool.</td>
</tr>
<tr>
<td>getBusyOption() on page 13-206</td>
<td>Returns the behavior of the stateless connection pool when all the connections in the pool are busy and the number of connections have reached maximum</td>
</tr>
<tr>
<td>getConnection() on page 13-206</td>
<td>Returns a pointer to the Connection object.</td>
</tr>
<tr>
<td>getIncrConnections() on page 13-208</td>
<td>Returns the number of incremental connections in the connection pool.</td>
</tr>
<tr>
<td>getMaxConnections() on page 13-208</td>
<td>Returns the maximum number of connections in the connection pool.</td>
</tr>
<tr>
<td>getMinConnections() on page 13-208</td>
<td>Returns the minimum number of connections in the connection pool.</td>
</tr>
</tbody>
</table>
getAnyTaggedConnection()

Returns a pointer to the connection object, without the restriction of a matching tag. This method works in an environment with enabled database resident connection pooling.

During the execution of this call, the pool is first searched based on the tag provided. If a connection with the specified tag exists, it is returned to the user. If a matching connection is not available, an appropriately authenticated untagged connection (with a NULL tag) is returned. In cases where an undated connection is not free, an appropriately authenticated connection with a different tag is returned.

A getTag() call to the Connection verifies that the connection tag is received.

**Syntax**

```c++
Connection *getAnyTaggedConnection(
    string &tag="")=0;

Connection* getAnyTaggedConnection(
    const UString &tag)=0;

Connection *getAnyTaggedConnection(
    const string &userName,
    const string &password,
    const string &tag="")=0;
```

**Description**

Returns a pointer to the connection object from a homogeneous stateless connection pool, without the restriction of a matching tag; string support.

Returns a pointer to the connection object from a homogeneous stateless connection pool, without the restriction of a matching tag; UString support.

Returns a pointer to the connection object from a heterogeneous stateless connection pool, without the restriction of a matching tag; string support.

---

**Table 13–42 (Cont.) Summary of StatelessConnectionPool Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getOpenConnections() on page 13-208</td>
<td>Returns the number of open connections in the connection pool.</td>
</tr>
<tr>
<td>getPoolName() on page 13-208</td>
<td>Returns the name of the connection pool.</td>
</tr>
<tr>
<td>getProxyConnection() on page 13-208</td>
<td>Returns a proxy connection from a connection pool.</td>
</tr>
<tr>
<td>getTimeOut() on page 13-210</td>
<td>Returns the timeout period of a connection in the connection pool.</td>
</tr>
<tr>
<td>releaseConnection() on page 13-210</td>
<td>Releases the connection back to the pool with an optional tag.</td>
</tr>
<tr>
<td>setBusyOption() on page 13-211</td>
<td>Specifies the behavior of the stateless connection pool when:</td>
</tr>
<tr>
<td></td>
<td>■ all the connections in the pool are busy, and</td>
</tr>
<tr>
<td></td>
<td>■ the number of connections have reached maximum.</td>
</tr>
<tr>
<td>setPoolSize() on page 13-211</td>
<td>Sets the maximum, minimum, and incremental number of pooled connections for the connection pool.</td>
</tr>
<tr>
<td>setTimeOut() on page 13-211</td>
<td>Sets the timeout period of a connection in the connection pool.</td>
</tr>
<tr>
<td>terminateConnection() on page 13-212</td>
<td>Closes the connection and remove it from the pool.</td>
</tr>
</tbody>
</table>
**getAnyTaggedProxyConnection()**

Returns a proxy connection from a connection pool.

This method works in an environment with enabled database resident connection pooling.

During the execution of this call, the pool is first searched based on the tag provided. If a connection with the specified tag exists, it is returned to the user. If a matching connection is not available, an appropriately authenticated connection with a different tag is returned. In cases where an undated connection is not free, an appropriately authenticated connection with a different tag is returned.

Restrictions for matching the tag may be removed by passing an empty tag argument parameter.

A getTag() call to the connection verifies the connection tag received.
**getBusyConnections()**

Returns the number of busy connections in the connection pool. When using database resident connection pooling, this is the number of persistent connections to the Connection Broker.

**Syntax**

```cpp
unsigned int getBusyConnections() const = 0;
```

**getBusyOption()**

Returns the behavior of the stateless connection pool when all the connections in the pool are busy, and when the number of connections have reached maximum. The return values are defined for BusyOption in Table 13–41 on page 13-203.

**Syntax**

```cpp
BusyOption getBusyOption() = 0;
```

**getConnection()**

Returns a pointer to the connection object of a StatelessConnectionPool. This method works in an environment with enabled database resident connection pooling.

**Syntax**

```cpp
Connection* getConnection() = 0;
```

---

**getAnyTaggedProxyConnection()**

Get a proxy connection within role specifications from the connection pool; UString support.

**Syntax**

```cpp
Connection* getAnyTaggedProxyConnection(
    const UString &name,
    const UString &tag,
    Connection::ProxyType proxyType = Connection::PROXY_DEFAULT);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The username.</td>
</tr>
<tr>
<td>roles</td>
<td>The roles to activate on the database server</td>
</tr>
<tr>
<td>numRoles</td>
<td>The number of roles to activate on the database server</td>
</tr>
<tr>
<td>tag</td>
<td>User defined tag associated with the connection.</td>
</tr>
<tr>
<td>proxyType</td>
<td>The type of proxy authentication to perform; ProxyType is defined in Table 13–11 on page 13-49.</td>
</tr>
</tbody>
</table>
StatelessConnectionPool Class

### OCCI Application Programming Interface

#### 13-207

Connection* getConnection(
    const UString &tag)=0;

Returns an authenticated connection, with a connection pool username and password; UString support.

Connection *getConnection(
    const string &userName,
    const string &password,
    const string &tag="")=0;

Returns a pointer to the connection object from a heterogeneous stateless connection pool; string support.

Connection* getConnection(
    const UString &userName,
    const UString &password,
    const UString &tag)=0;

Returns a pointer to the connection object from a heterogeneous stateless connection pool; UString support.

Connection *getConnection(
    const string &userName,
    const string &password,
    const string &tag=")=0;

Returns a pointer to the connection object from a heterogeneous stateless connection pool; string support.

Connection* getConnection(
    const UString &connectionClass,
    const Connection::Purity &purity)=0;

Returns a pointer to the connection object from a database resident connection pool; string support.

Connection *getConnection(
    const string &connectionClass,
    const Connection::Purity &purity)=0;

Returns a pointer to the connection object from a database resident connection pool; UString support.

Connection* getConnection(
    const string &connectionClass,
    const Connection::Purity &purity,
    const string &tag)=0;

Returns a tagged connection object from a database resident connection pool; string support.

Connection *getConnection(
    const string &connectionClass,
    const Connection::Purity &purity,
    const UString &tag)=0;

Returns a tagged connection object from a database resident connection pool; UString support.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userName</td>
<td>The database username.</td>
</tr>
<tr>
<td>password</td>
<td>The database password.</td>
</tr>
<tr>
<td>tag</td>
<td>The user defined tag associated with the connection. During the call, the pool is first searched based on the tag provided. If a connection with the specified tag exists it is returned; otherwise a new connection is created and returned.</td>
</tr>
<tr>
<td>connectionClass</td>
<td>The class of connection used by database resident connection pool.</td>
</tr>
<tr>
<td>purity</td>
<td>The purity of connection used by database resident connection pool; either SELF or NEW.</td>
</tr>
</tbody>
</table>
getIncrConnections()

Returns the number of incremental connections in the connection pool. This call is useful only in cases of homogeneous connection pools. When using database resident connection pooling, this is the number of persistent connections to the Connection Broker.

**Syntax**

```cpp
unsigned int getIncrConnections() const=0;
```

getMaxConnections()

Returns the maximum number of connections in the connection pool. When using database resident connection pooling, this is the number of persistent connections to the Connection Broker.

**Syntax**

```cpp
unsigned int getMaxConnections() const=0;
```

getMinConnections()

Returns the minimum number of connections in the connection pool. When using database resident connection pooling, this is the number of persistent connections to the Connection Broker.

**Syntax**

```cpp
unsigned int getMinConnections() const=0;
```

getOpenConnections()

Returns the number of open connections in the connection pool. When using database resident connection pooling, this is the number of persistent connections to the Connection Broker.

**Syntax**

```cpp
unsigned int getOpenConnections() const=0;
```

getPoolName()

Returns the name of the connection pool.

**Syntax**

```cpp
string getPoolName() const=0;
```

getProxyConnection()

Returns a proxy connection from a connection pool.
This method works in an environment with enabled database resident connection pooling.

### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| `Connection *getProxyConnection(
  const string &userName,
  string roles[],
  unsigned int numRoles,
  const string& tag="",
  Connection::ProxyType
    proxyType=Connection::PROXY_DEFAULT)=0;` | Get a proxy connection with role specifications from a connection pool; support for roles and string support. |
| `Connection* getProxyConnection(
  const UString &userName,
  UString roles[],
  unsigned int numRoles,
  const UString &tag,
  Connection::ProxyType
    proxyType = Connection::PROXY_DEFAULT);` | Get a proxy connection with role specifications from a connection pool; support for roles and UString support. |
| `Connection *getProxyConnection(
  const string &userName,
  const string &connectionClass,
  const Connection::Purity &purity)=0;` | Get a proxy connection from a database resident connection pool; string support. |
| `Connection* getProxyConnection(
  const UString &userName,
  const UString &connectionClass,
  const Connection::Purity &purity)=0;` | Get a proxy connection from a database resident connection pool; UString support. |
| `Connection *getProxyConnection(
  const string &userName,
  string roles[],
  unsigned int numRoles,
  const string &connectionClass,
  const Connection::Purity &purity)=0;` | Get a proxy connection with role specifications from a connection pool; support for roles and database resident connection pooling; string support. |
| `Connection* getProxyConnection(
  const UString &userName,
  UString roles[],
  unsigned int numRoles,
  const UString &connectionClass,
  const Connection::Purity &purity)=0;` | Get a proxy connection with role specifications from a connection pool; support for roles and database resident connection pooling; UString support. |
| `Connection *getProxyConnection(
  const string &userName,
  const string &tag="",
  Connection::ProxyType
    proxyType=Connection::PROXY_DEFAULT)=0;` | Get a proxy connection without role specifications from a connection pool; string support. |
| `Connection* getProxyConnection(
  const UString &userName,
  const UString &tag,
  Connection::ProxyType
    proxyType = Connection::PROXY_DEFAULT)` | Get a proxy connection without role specifications from a connection pool; UString support. |

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userName</td>
<td>The user name.</td>
</tr>
<tr>
<td>roles</td>
<td>The roles to activate on the database server.</td>
</tr>
</tbody>
</table>
getStmtCacheSize()

Retrieves the size of the statement cache.

**Syntax**

```cpp
unsigned int getStmtCacheSize() const = 0;
```

getTimeOut()

Returns the timeout period of a connection in the connection pool.

**Syntax**

```cpp
unsigned int getTimeOut() const = 0;
```

releaseConnection()

Releases the connection back to the pool with an optional tag.
This method works in an environment with enabled database resident connection pooling.

**Syntax**

```cpp
void releaseConnection(
    Connection *connection,
    const string& tag="");

void releaseConnection(
    Connection *connection,
    const UString &tag);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numRoles</td>
<td>The number of roles to activate on the database server.</td>
</tr>
<tr>
<td>tag</td>
<td>The user defined tag associated with the connection. During the execution of this call, the pool is first searched based on the tag provided. If a connection with the specified tag exists it is returned; otherwise, a new connection is created and returned.</td>
</tr>
<tr>
<td>proxyType</td>
<td>The type of proxy authentication to perform; ProxyType is defined in Table 13–11 on page 13-49.</td>
</tr>
<tr>
<td>connectionClass</td>
<td>The class of connection used by database resident connection pool.</td>
</tr>
<tr>
<td>purity</td>
<td>The purity of connection used by database resident connection pool; either SELF or NEW.</td>
</tr>
</tbody>
</table>

Parameter Description

- **connection**
  - The connection to be released.

- **tag**
  - The user defined tag associated with the connection. The default of this parameter is "", which untags the connection.
**setBusyOption()**

Specifies the behavior of the stateless connection pool when all the connections in the pool are busy, and when the number of connections have reached maximum.

**Syntax**

```c
void setBusyOption(
    BusyOption busyOption)=0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>busyOption</td>
<td>Valid values are defined in BusyOption in Table 13-41 on page 13-203.</td>
</tr>
</tbody>
</table>

**setPoolSize()**

Sets the maximum, minimum, and incremental number of pooled connections for the connection pool.

**Syntax**

```c
void setPoolSize(
    unsigned int maxConn=1,
    unsigned int minConn=0,
    unsigned int incrConn=1)=0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxConn</td>
<td>The maximum number of connections in the connection pool.</td>
</tr>
<tr>
<td>minConn</td>
<td>The minimum number of connections, in homogeneous pools only.</td>
</tr>
<tr>
<td>incrConn</td>
<td>The incremental number of connections, in homogeneous pools only.</td>
</tr>
</tbody>
</table>

**setTimeOut()**

Sets the time out period of a connection in the connection pool. OCCI terminates any connections related to this connection pool that have been idle for longer than the timeout period specified.

If this attribute is not set, the least recently used connection is timed out when pool space is required. Oracle only checks for timed out connections when it releases a connection back to the pool.

**Syntax**

```c
void setTimeOut(
    unsigned int connTimeOut=0)=0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connTimeOut</td>
<td>The time out period, given in seconds.</td>
</tr>
</tbody>
</table>
setStmtCacheSize()

Enables or disables statement caching. A nonzero value enables statement caching, with a cache of specified size. A zero value disables caching.

If the user changes the cache size of individual connections and subsequently returns the connection back to the pool with a tag, the cache size does not revert to the one set for the pool. If the connection is untagged, the cache size is reset to equal the cache size specified for the pool.

**Syntax**

```cpp
void setStmtCacheSize(
    unsigned int cacheSize)=0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacheSize</td>
<td>The size of the statement cache</td>
</tr>
</tbody>
</table>

terminateConnection()

Closes the connection and removes it from the pool.

This method works in an environment with enabled database resident connection pooling.

**Syntax**

```cpp
void terminateConnection(
    Connection *connection)=0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>The connection to be terminated</td>
</tr>
</tbody>
</table>
Statement Class

A `Statement` object is used for executing SQL statements. The statement may be a query returning result set, or a non-query statement returning an update count. Non-query SQL can be insert, update, or delete statements. Non-query SQL statements can also be DML statements (such as create, grant, and so on) or stored procedure calls.

A query, insert / update / delete, or stored procedure call statements may have `IN` bind parameters, while a stored procedure call statement may have either `OUT` bind parameters or bind parameters that are both `IN` and `OUT`, referred to as `IN/OUT` parameters.

The `Statement` class methods are divided into three categories:

- Statement methods applicable to all statements
- Methods applicable to prepared statements with `IN` bind parameters
- Methods applicable to callable statements with `OUT` or `IN/OUT` bind parameters.

### Table 13-43 Enumerated Values used by the Statement Class

<table>
<thead>
<tr>
<th>Attribute Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NEEDS_STREAM_DATA</code></td>
<td>indicates that output Streams must be written for the streamed IN bind parameters. If there are multiple streamed parameters, call the <code>getCurrentStreamParam()</code> method to find out the bind parameter that needs the stream. If the statement is executed iteratively, call <code>getCurrentIteration()</code> to find the iteration for the stream that must to be written.</td>
</tr>
<tr>
<td><code>PREPARED</code></td>
<td>indicates that the <code>Statement</code> is set to a query.</td>
</tr>
<tr>
<td><code>RESULT_SET_AVAILABLE</code></td>
<td>indicates that the <code>getResultSet()</code> method must be called to get the result set.</td>
</tr>
<tr>
<td><code>STREAM_DATA_AVAILABLE</code></td>
<td>indicates that the input Streams must be read for the streamed OUT bind parameters. If there are multiple streamed parameters, call the <code>getCurrentStreamParam()</code> method to find out the bind parameter that needs the stream. If the statement is executed iteratively, call <code>getCurrentIteration()</code> to find the iteration for the stream that must be read.</td>
</tr>
<tr>
<td><code>UPREPARED</code></td>
<td>indicates that the <code>Statement</code> object is not set to a query.</td>
</tr>
<tr>
<td><code>UPDATE_COUNT_AVAILABLE</code></td>
<td>indicates that the <code>getUpdateCount()</code> method must be called to find out the update count.</td>
</tr>
</tbody>
</table>

### Table 13-44 Statement Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addIteration()</code></td>
<td>Adds an iteration for execution.</td>
</tr>
<tr>
<td><code>closeResultSet()</code></td>
<td>Immediately releases a result set's database and OCCI resources instead of waiting for automatic release.</td>
</tr>
<tr>
<td><code>closeStream()</code></td>
<td>Closes the stream specified by the parameter <code>stream</code>.</td>
</tr>
<tr>
<td><code>disableCaching()</code></td>
<td>Disables statement caching.</td>
</tr>
<tr>
<td><code>execute()</code></td>
<td>Runs the SQL statement.</td>
</tr>
<tr>
<td><code>executeArrayUpdate()</code></td>
<td>Runs insert, update, and delete statements that use only the <code>setDataBuffer()</code> or stream interface for bind parameters.</td>
</tr>
</tbody>
</table>
Table 13–44  (Cont.) Statement Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>executeQuery()</code> on page 13-219</td>
<td>Runs a SQL statement that returns a single ResultSet.</td>
</tr>
<tr>
<td><code>executeUpdate()</code> on page 13-219</td>
<td>Runs a SQL statement that does not return a ResultSet.</td>
</tr>
<tr>
<td><code>getAutoCommit()</code> on page 13-219</td>
<td>Returns the current auto-commit state.</td>
</tr>
<tr>
<td><code>getBatchErrorMode()</code> on page 13-219</td>
<td>Returns the state of the batch error mode.</td>
</tr>
<tr>
<td><code>getBDouble()</code> on page 13-220</td>
<td>Returns the value of an IEEE754 DOUBLE as a BDouble object.</td>
</tr>
<tr>
<td><code>getBfile()</code> on page 13-220</td>
<td>Returns the value of a BFILE as a Bfile object.</td>
</tr>
<tr>
<td><code>getBFloat()</code> on page 13-220</td>
<td>Returns the value of an IEEE754 FLOAT as a BFloat object.</td>
</tr>
<tr>
<td><code>getBlob()</code> on page 13-220</td>
<td>Returns the value of a BLOB as a Blob object.</td>
</tr>
<tr>
<td><code>getBytes()</code> on page 13-221</td>
<td>Returns the value of a SQL BINARY or VARBINARY parameter as Bytes.</td>
</tr>
<tr>
<td><code>getCharSet()</code> on page 13-221</td>
<td>Returns the characterset that is in effect for the specified parameter, as a string.</td>
</tr>
<tr>
<td><code>getCharSetUString()</code> on page 13-221</td>
<td>Returns the characterset that is in effect for the specified parameter, as a UString.</td>
</tr>
<tr>
<td><code>getBlob()</code> on page 13-221</td>
<td>Returns the value of a CLOB as a Clob object.</td>
</tr>
<tr>
<td><code>getConnection()</code> on page 13-222</td>
<td>Returns the connection from which the Statement object was instantiated.</td>
</tr>
<tr>
<td><code>getCurrentIteration()</code> on page 13-222</td>
<td>Returns the iteration number of the current iteration that is being processed.</td>
</tr>
<tr>
<td><code>getCurrentStreamIteration()</code> on page 13-222</td>
<td>Returns the current iteration for which stream data is to be read or written.</td>
</tr>
<tr>
<td><code>getCurrentStreamParam()</code> on page 13-222</td>
<td>Returns the parameter index of the current output Stream that must be read or written.</td>
</tr>
<tr>
<td><code>getCursor()</code> on page 13-222</td>
<td>Returns the REF CURSOR value of an OUT parameter as a ResultSet.</td>
</tr>
<tr>
<td><code>getDatabaseNCHARParam()</code> on page 13-223</td>
<td>Returns whether data is in NCHAR character set.</td>
</tr>
<tr>
<td><code>getDouble()</code> on page 13-223</td>
<td>Returns the value of a parameter as a C++ double.</td>
</tr>
<tr>
<td><code>getIntervalDS()</code> on page 13-224</td>
<td>Returns the value of a parameter as an IntervalDS object.</td>
</tr>
<tr>
<td><code>getIntervalYM()</code> on page 13-224</td>
<td>Returns the value of a parameter as an IntervalYM object.</td>
</tr>
<tr>
<td><code>getMaxIterations()</code> on page 13–224</td>
<td>Returns the current limit on maximum number of iterations.</td>
</tr>
<tr>
<td><code>getMaxParamSize()</code> on page 13-225</td>
<td>Returns the current max parameter size limit.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getNumber() on page 13-225</td>
<td>Returns the value of a parameter as a Number object.</td>
</tr>
<tr>
<td>getObject() on page 13-225</td>
<td>Returns the value of a parameter as a PObject.</td>
</tr>
<tr>
<td>getOCIStatement() on page 13-225</td>
<td>Returns the OCI statement handle associated with the Statement.</td>
</tr>
<tr>
<td>getRef() on page 13-226</td>
<td>Returns the value of a REF parameter as RefAny</td>
</tr>
<tr>
<td>getResultSet() on page 13-226</td>
<td>Returns the current result as a ResultSet.</td>
</tr>
<tr>
<td>getRowid() on page 13-226</td>
<td>Returns the row id parameter value as a Bytes object.</td>
</tr>
<tr>
<td>getSQL() on page 13-226</td>
<td>Returns the current SQL string associated with the Statement object.</td>
</tr>
<tr>
<td>getSQLUString() on page 13-226</td>
<td>Returns the current SQL string associated with the Statement object; globalization enabled.</td>
</tr>
<tr>
<td>getStream() on page 13-227</td>
<td>Returns the value of the parameter as a stream.</td>
</tr>
<tr>
<td>getString() on page 13-227</td>
<td>Returns the value of the parameter as a string.</td>
</tr>
<tr>
<td>getTimestamp() on page 13-227</td>
<td>Returns the value of the parameter as a Timestamp object</td>
</tr>
<tr>
<td>getUInt() on page 13-227</td>
<td>Returns the value of the parameter as a C++ unsigned int</td>
</tr>
<tr>
<td>getUpdateCount() on page 13-228</td>
<td>Returns the current result as an update count for non-query statements.</td>
</tr>
<tr>
<td>getUString() on page 13-228</td>
<td>Returns the value of a UString.</td>
</tr>
<tr>
<td>getVector() on page 13-228</td>
<td>Returns the specified parameter as a vector.</td>
</tr>
<tr>
<td>getVectorOfRefs() on page 13-230</td>
<td>Returns the column in the current position as a vector of REFS.</td>
</tr>
<tr>
<td>isNull() on page 13-230</td>
<td>Checks whether the parameter is NULL.</td>
</tr>
<tr>
<td>isTruncated() on page 13-231</td>
<td>Checks whether the value is truncated.</td>
</tr>
<tr>
<td>preTruncationLength() on page 13-231</td>
<td>Returns the actual length of the parameter before truncation.</td>
</tr>
<tr>
<td>registerOutParam() on page 13-231</td>
<td>Registers the type and max size of the OUT parameter.</td>
</tr>
<tr>
<td>setAutoCommit() on page 13-232</td>
<td>Specifies auto commit mode.</td>
</tr>
<tr>
<td>setBatchErrorMode() on page 13-232</td>
<td>Enables or disables the batch error processing mode.</td>
</tr>
<tr>
<td>setBDouble() on page 13-233</td>
<td>Sets a parameter to an IEEE double value.</td>
</tr>
<tr>
<td>setBFile() on page 13-233</td>
<td>Sets a parameter to a Bfile value.</td>
</tr>
<tr>
<td>setBFloat() on page 13-233</td>
<td>Sets a parameter to an IEEE float value.</td>
</tr>
<tr>
<td>setBinaryStreamMode() on page 13-234</td>
<td>Specifies that a column is to be returned as a binary stream.</td>
</tr>
<tr>
<td>setBlob() on page 13-234</td>
<td>Sets a parameter to a Blob value.</td>
</tr>
<tr>
<td>setBytes() on page 13-234</td>
<td>Sets a parameter to a Bytes array.</td>
</tr>
<tr>
<td>setCharacterSet() on page 13-235</td>
<td>Specifies that a column is to be returned as a character stream.</td>
</tr>
<tr>
<td>setCharSetUString() on page 13-235</td>
<td>Specifies the character set as a string.</td>
</tr>
</tbody>
</table>
Table 13–44  (Cont.) Statement Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setClob() on page 13-236</td>
<td>Sets a parameter to a Clob value.</td>
</tr>
<tr>
<td>setDate() on page 13-236</td>
<td>Sets a parameter to a Date value.</td>
</tr>
<tr>
<td>setDatabaseNCHARParam() on page 13-236</td>
<td>Sets to true if the data is to be in the NCHAR character set of the database; set to false to restore the default.</td>
</tr>
<tr>
<td>setDBuffer() on page 13-237</td>
<td>Specifies a data buffer where data would be available for reading or writing.</td>
</tr>
<tr>
<td>setDBufferArray() on page 13-238</td>
<td>Specifies an array of data buffers where data would be available for reading or writing.</td>
</tr>
<tr>
<td>setDouble() on page 13-239</td>
<td>Sets a parameter to a C++ double value.</td>
</tr>
<tr>
<td>setErrorOnNull() on page 13-239</td>
<td>Enables or disables exceptions for reading of NULL values.</td>
</tr>
<tr>
<td>setErrorOnTruncate() on page 13-239</td>
<td>Enables or disables exception when truncation occurs.</td>
</tr>
<tr>
<td>setFloat() on page 13-240</td>
<td>Sets a parameter to a C++ float value.</td>
</tr>
<tr>
<td>setInt() on page 13-240</td>
<td>Sets a parameter to a C++ int value.</td>
</tr>
<tr>
<td>setIntervalDS() on page 13-240</td>
<td>Sets a parameter to an IntervalDS value.</td>
</tr>
<tr>
<td>setIntervalYM() on page 13-240</td>
<td>Sets a parameter to an IntervalYM value.</td>
</tr>
<tr>
<td>setMaxIterations() on page 13-241</td>
<td>Sets the maximum number of invocations that are made for the DML statement.</td>
</tr>
<tr>
<td>setMaxParamSize() on page 13-241</td>
<td>Sets the maximum amount of data that can sent or returned from the parameter.</td>
</tr>
<tr>
<td>setNull() on page 13-242</td>
<td>Sets a parameter to SQL NULL.</td>
</tr>
<tr>
<td>setNumber() on page 13-242</td>
<td>Sets a parameter to a Number value.</td>
</tr>
<tr>
<td>setObject() on page 13-242</td>
<td>Sets the value of a parameter using an object.</td>
</tr>
<tr>
<td>setPrefetchMemorySize() on page 13-243</td>
<td>Sets the amount of memory that is used internally by OCCI to store data fetched during each round trip to the server.</td>
</tr>
<tr>
<td>setPrefetchRowCount() on page 13-243</td>
<td>Sets the number of rows that are fetched internally by OCCI during each round trip to the server.</td>
</tr>
<tr>
<td>setRef() on page 13-243</td>
<td>Sets the value of a parameter to a reference.</td>
</tr>
<tr>
<td>setRowid() on page 13-244</td>
<td>Sets a row id bytes array for a bind position.</td>
</tr>
<tr>
<td>setSQL() on page 13-244</td>
<td>Associates new SQL string with Statement object.</td>
</tr>
<tr>
<td>setSQLUString() on page 13-245</td>
<td>Associates new SQL string with Statement object; globalization enabled.</td>
</tr>
<tr>
<td>setString() on page 13-245</td>
<td>Sets a parameter for a specified index.</td>
</tr>
<tr>
<td>setTimestamp() on page 13-245</td>
<td>Sets a parameter to a Timestamp value.</td>
</tr>
<tr>
<td>setUInt() on page 13-245</td>
<td>Sets a parameter to a C++ unsigned int value.</td>
</tr>
<tr>
<td>setUString() on page 13-246</td>
<td>Sets a parameter for a specified index; globalization enabled.</td>
</tr>
<tr>
<td>setVector() on page 13-246</td>
<td>Sets a parameter to a vector of unsigned int.</td>
</tr>
<tr>
<td>setVectorOfRefs() on page 13-252</td>
<td>Sets a parameter to a vector; is necessary when the type is a collection of REFS.</td>
</tr>
</tbody>
</table>
addIteration()

After specifying set parameters, an iteration is added for execution.

Syntax

void addIteration();

closeResultSet()

Immediately releases the specified resultSet's database and OCCI resources when the result set is not needed.

Syntax

void closeResultSet(
    ResultSet *resultSet);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resultSet</td>
<td>The result set to be closed; may be a result of resultSet(), executeQuery(), or getCursor() calls on the current statement, or by a getCursor() call of another result set on the same statement.</td>
</tr>
</tbody>
</table>

closeStream()

Closes the stream specified by the parameter stream.

Syntax

void closeStream(
    Stream *stream); 

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>The stream to be closed.</td>
</tr>
</tbody>
</table>

disableCaching()

Disables statement caching. Used if a user wants to destroy a statement instead of caching it. Effective only if statement caching is enabled.

Syntax

void disableCaching();
**execute()**

Executes an SQL statement that may return either a result set or an update count. The statement may have read-able streams which may have to be written, in which case the results of the execution may not be readily available. The returned value `Status` is defined in Table 13–43 on page 13-213.

If output streams are used for `OUT` bind variables, they must be completely read in order. The `getCurrentStreamParam()` method would indicate which stream must be read. Similarly, `getCurrentIteration()` would indicate the iteration for which data is available.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status execute(const string &amp;sql=&quot;&quot;);</td>
<td>Executes the SQL Statement.</td>
</tr>
<tr>
<td>Status execute(const UString &amp;sql);</td>
<td>Executes the SQL Statement; globalization enabled.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td>The SQL statement to be executed. This can be <code>NULL</code> if the <code>executeArrayUpdate()</code> method was used to associate the <code>sql</code> with the statement.</td>
</tr>
</tbody>
</table>

**executeArrayUpdate()**

Executes insert/update/delete statements which use only the `setDataBuffer()` or stream interface for bind parameters. The bind parameters must be arrays of size `arrayLength` parameter. The statement may have writeable streams which may have to be written. The returned value `Status` is defined in Table 13–43 on page 13-213.

If output streams are used for `OUT` bind variables, they must be completely read in order. The `getCurrentStreamParam()` method would indicate which stream must be read. Similarly, `getCurrentIteration()` would indicate the iteration for which data is available.

Note that you cannot perform array executes for queries or callable statements.

**Syntax**

```cpp
Status executeArrayUpdate(
    unsigned int arrayLength);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrayLength</td>
<td>The number of elements provided in each buffer of bind variables.</td>
</tr>
</tbody>
</table>
executeQuery()

Runs a SQL statement that returns a ResultSet. Should not be called for a statement which is not a query, has streamed parameters. Returns a ResultSet that contains the data produced by the query.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| ResultSet* executeQuery( 
  const string &sql=""); | Executes the SQL Statement that returns a ResultSet. |
| ResultSet* executeQuery( 
  const UString &sql); | Executes the SQL Statement that returns a ResultSet; globalization enabled. |

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td>The SQL statement to be executed. This can be NULL if the executeArrayUpdate() method was used to associate the sql with the statement.</td>
</tr>
</tbody>
</table>

executeUpdate()

Executes a non-query statement such as a SQL INSERT, UPDATE, DELETE statement, a DDL statement such as CREATE/ALTER and so on, or a stored procedure call. Returns either the row count for INSERT, UPDATE or DELETE or 0 for SQL statements that return nothing.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| unsigned int executeUpdate( 
  const string &sql=""); | Executes a non-query statement. |
| unsigned int executeUpdate( 
  const UString &sql); | Executes a non-query statement; globalization enabled. |

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td>The SQL statement to be executed. This can be NULL if the executeArrayUpdate() method was used to associate the sql with the statement.</td>
</tr>
</tbody>
</table>

getAutoCommit()

Returns the current auto-commit state.

Syntax

bool getAutoCommit() const;

getBatchErrorMode()

Returns the state of the batch error mode; TRUE if the batch error mode is enabled, FALSE otherwise.
**getBDouble()**

_Returns the value of an IEEE754 DOUBLE column, which has been defined as an **OUT** bind. If the value is SQL **NULL**, the result is 0._

**Syntax**

```cpp
bool getBatchErrorMode() const;
```

```cpp
BDouble getBDouble(  
    unsigned int paramIndex) = 0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getBfile()**

_Returns the value of a BFILE parameter as a Bfile object._

**Syntax**

```cpp
Bfile getBfile(  
    unsigned int paramIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getBFloat()**

_Gets the value of an IEEE754 FLOAT column, which has been defined as an **OUT** bind. If the value is SQL **NULL**, the result is 0._

**Syntax**

```cpp
BFloat getBFloat(  
    unsigned int paramIndex) = 0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getBlob()**

_Returns the value of a BLOB parameter as a Blob._

**Syntax**

```cpp
Blob getBlob(  
    unsigned int paramIndex);
```
getBytes()

Returns the value of an SQL BINARY or VARBINARY parameter as Bytes; if the value is SQL NULL, the result is NULL.

**Syntax**

```cpp
Bytes getBytes(
    unsigned int paramIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getCharSet()

Returns the character set that is in effect for the specified parameter, as a string.

**Syntax**

```cpp
string getCharSet(
    unsigned int paramIndex) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getCharSetUString()

Returns the character set that is in effect for the specified parameter, as a UString.

**Syntax**

```cpp
UString getCharSetUString(
    unsigned int paramIndex) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getclob()

Get the value of a CLOB parameter as a Clob. Returns the parameter value.

**Syntax**

```cpp
Clob getClob(
    unsigned int paramIndex);
```
getConnection()

Returns the connection from which the Statement object was instantiated.

Syntax

Connection* getConnection() const;

gGetCurrentIteration()

If the prepared statement has any output Streams, this method returns the current iteration of the statement that is being processed by OCCI. If this method is called after all the invocations in the set of iterations has been processed, it returns 0. Returns the iteration number of the current iteration that is being processed. The first iteration is numbered 1 and so on. If the statement has finished execution, a 0 is returned.

Syntax

unsigned int getCurrentIteration() const;

gGetCurrentStreamIteration()

Returns the current parameter stream for which data is available.

Syntax

unsigned int getCurrentStreamIteration() const;

gGetCurrentStreamParam()

Returns the parameter index of the current output Stream parameter that must be written. If the prepared statement has any output Stream parameters, this method returns the parameter index of the current output Stream that must be written. If no output Stream must be written, or there are no output Stream parameters in the prepared statement, this method returns 0.

Syntax

unsigned int getCurrentStreamParam() const;

gCursor()

Gets the REF CURSOR value of an OUT parameter as a ResultSet. Data can be fetched from this result set. The OUT parameter must be registered as CURSOR with the registerOutParam() method. Returns a ResultSet for the OUT parameter value.

Note that if there are multiple REF CURSORS being returned due to a batched call, data from each cursor must be completely fetched before retrieving the next REF CURSOR and starting fetch on it.
Syntax

ResultSet * getCursor(
    unsigned int paramIndex);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getDatabaseNCHARParam()

Returns whether data is in NCHAR character set or not.

Syntax

bool getDatabaseNCHARParam(
    unsigned int paramIndex) const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getDate()

Get the value of a SQL DATE parameter as a Date object. Returns the parameter value; if the value is SQL NULL, the result is NULL.

Syntax

Date getDate(
    unsigned int paramIndex) const;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getDouble()

Get the value of a DOUBLE parameter as a C++ double. Returns the parameter value; if the value is SQL NULL, the result is 0.

Syntax

double getDouble(
    unsigned int paramIndex);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getFloat()

Get the value of a FLOAT parameter as a C++ float. Returns the parameter value; if the value is SQL NULL, the result is 0.
**getInt()**

Syntax

```c
float getFloat(
    unsigned int paramIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getInt()**

Get the value of an INTEGER parameter as a C++ int. Returns the parameter value; if the value is SQL NULL, the result is 0.

Syntax

```c
unsigned int getInt(
    unsigned int paramIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getIntervalDS()**

Get the value of a parameter as a IntervalDS object.

Syntax

```c
IntervalDS getIntervalDS(
    unsigned int paramIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getIntervalYM()**

Get the value of a parameter as a IntervalYM object.

Syntax

```c
IntervalYM getIntervalYM(
    unsigned int paramIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getMaxIterations()**

Gets the current limit on maximum number of iterations. Default is 1. Returns the current maximum number of iterations.
Statement Class

Syntax

unsigned int getMaxIterations() const;

getMaxParamSize()

The maxParamSize limit (in bytes) is the maximum amount of data sent or returned for any parameter value; it only applies to character and binary types. If the limit is exceeded, the excess data is silently discarded. Returns the current max parameter size limit.

Syntax

unsigned int getMaxParamSize(
    unsigned int paramIndex) const;

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

getNumber()

Gets the value of a NUMERIC parameter as a Number object. Returns the parameter value; if the value is SQL NULL, the result is NULL.

Syntax

Number getNumber(
    unsigned int paramIndex);

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

gGetObject()

Gets the value of a parameter as a PObject. This method returns an PObject whose type corresponds to the SQL type that was registered for this parameter using registerOutParam(). Returns A PObject holding the OUT parameter value.

This method may be used to read database-specific, abstract data types.

Syntax

PObject * getobject(
    unsigned int paramIndex);

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

ggetOCIStatement()

Get the OCI statement handle associated with the Statement.
**getRef()**

Get the value of a REF parameter as RefAny. Returns the parameter value.

**Syntax**

```cpp
RefAny getRef(unsigned int paramIndex);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getResultSet()**

Returns the current result as a ResultSet.

**Syntax**

```cpp
ResultSet * getResultSet();
```

**getRowid()**

Get the rowid parameter value as a Bytes.

**Syntax**

```cpp
BytesgetRowid(unsigned int paramIndex);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getSQL()**

Returns the current SQL string associated with the Statement object.

**Syntax**

```cpp
string getSQL() const;
```

**getSQLUString()**

Returns the current SQL UString associated with the Statement object; globalization enabled.

**Syntax**

```cpp
UString getSQLUString() const;
```
**getStream()**

Returns the value of the parameter as a stream.

**Syntax**

```c
Stream * getStream(
    unsigned int paramIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getString()**

Get the value of a CHAR, VARCHAR, or LONGVARCHAR parameter as an string. Returns the parameter value; if the value is SQL NULL, the result is empty string.

**Syntax**

```c
string getString(
    unsigned int paramIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getTimestamp()**

Get the value of a SQL TIMESTAMP parameter as a Timestamp object. Returns the parameter value; if the value is SQL NULL, the result is NULL.

**Syntax**

```c
Timestamp getTimestamp(
    unsigned int paramIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**getUInt()**

Get the value of a BIGINT parameter as a C++ unsigned int. Returns the parameter value; if the value is SQL NULL, the result is 0.

**Syntax**

```c
unsigned int getUInt(
    unsigned int paramIndex);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>
getUpdateCount()

Returns the current result as an update count.

Syntax

unsigned int getUpdateCount() const;

getUString()

Returns the value as a UString.

This method should be called only if the environment's character set is UTF16, or if setCharset() method has been called to explicitly retrieve UTF16 data.

Syntax

UString getUString(
    unsigned int paramIndex);

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

gerVector()

Returns the column in the current position as a vector. The column at the position, specified by index, should be a collection type (varray or nested table). The SQL type of the elements in the collection should be compatible with the type of the vector.

Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void getVector( Statement *stmt,</td>
<td>Used for vectors of UString Class; globalization enabled.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>std::vector&lt;UString&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>void getVector( Statement *stmt,</td>
<td>Used for BDouble vectors.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;BDouble&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>void getVector( Statement *stmt,</td>
<td>Used for vectors of Bfile Class.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;BFile&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>void getVector( Statement *stmt,</td>
<td>Used for BFloat vectors.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;BFloat&gt; &amp;vect);</td>
<td></td>
</tr>
<tr>
<td>void getVector( Statement *stmt,</td>
<td>Used for vectors of Blob Class.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>vector&lt;Blob&gt; &amp;vect);</td>
<td></td>
</tr>
</tbody>
</table>
**Syntax** | **Description**  
--- | ---  
`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<Clob> &vect);` |  
  **Used for Clob vectors.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<Date> &vect);` |  
  **Used for vectors of Date Class.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<double> &vect);` |  
  **Used for vectors of double Class.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<float> &vect);` |  
  **Used for vectors of float Class.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<int> &vect);` |  
  **Used for vectors of int Class.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<IntervalDS> &vect);` |  
  **Used for vectors of IntervalDS Class.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<IntervalYM> &vect);` |  
  **Used for vectors of IntervalYM Class.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<Number> &vect);` |  
  **Used for vectors of Number Class.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<RefAny> &vect);` |  
  **Used for vectors of RefAny Class.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<string> &vect);` |  
  **Used for vectors of string Class.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<T *> &vect);` |  
  **Intended for use on platforms where partial ordering of function templates is supported.**  

`void getVector(` |  
  `Statement *stmt,` |  
  `unsigned int paramIndex,` |  
  `vector<T> &vect);` |  
  **Intended for use on platforms where partial ordering of function templates is not supported, such as Windows NT and z/OS. For OUT binds.**
getVectorOfRefs()

This method returns the column in the current position as a vector of REFs. The column should be a collection type (varray or nested table) of REFs. Used with OUT binds.

Syntax

```cpp
void getVectorOfRefs(
    Statement *stmt,
    unsigned int colIndex,
    vector<Ref<T> > &vect);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmt</td>
<td>The statement.</td>
</tr>
<tr>
<td>colIndex</td>
<td>Column index; first column is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>vect</td>
<td>The reference to the vector (OUT parameter) into which the values should be retrieved.</td>
</tr>
</tbody>
</table>

isNull()

An OUT parameter may have the value of SQL NULL; isNull() reports whether the last value read has this special value. Note that you must first call getxxx() on a parameter to read its value and then call isNull() to see if the value was SQL NULL. Returns TRUE if the last parameter read was SQL NULL.

Syntax

```cpp
bool isNull()
```
unsigned int paramIndex) const;

**Parameter** | **Description**
---|---
paramIndex | Parameter index; first parameter is 1, second is 2, and so on.

**isTruncated()**

This method checks whether the value of the parameter is truncated. If the value of the parameter is truncated, then **TRUE** is returned; otherwise, **FALSE** is returned.

**Syntax**

```cpp
bool isTruncated(
    unsigned int paramIndex) const;
```

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**preTruncationLength()**

Returns the actual length of the parameter before truncation.

**Syntax**

```cpp
int preTruncationLength(
    unsigned int paramIndex) const;
```

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>

**registerOutParam()**

This method registers the type of each **out** parameter of a PL/SQL stored procedure. Before executing a PL/SQL stored procedure, you must explicitly call this method to register the type of each **out** parameter. This method should be called for **out** parameters only. Use the `setxxx()` method for **in/out** parameters.

- When reading the value of an **out** parameter, you must use the `getxxx()` method that corresponds to the parameter's registered SQL type. For example, use `getInt` or `getNumber` when `OCCIINT` or `OCCINumber` is the type specified.
- If a PL/SQL stored procedure has an **out** parameter of type `ROWID`, the type specified in this method should be `OCCISTRING`. The value of the **out** parameter can then be retrieved by calling the `getString()` method.
- If a PL/SQL stored procedure has an **in/out** parameter of type `ROWID`, call the methods `setString()` and `getString()` to set the type and retrieve the value of the **IN/OUT** parameter.
setAutoCommit()

A Statement can be in auto-commit mode. In this case any statement executed is also automatically committed. By default, the auto-commit mode is turned-off.

Syntax

```c++
void setAutoCommit(
    bool autoCommit);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoCommit</td>
<td>TRUE enables auto-commit; FALSE disables auto-commit.</td>
</tr>
</tbody>
</table>

setBatchErrorMode()

Enables or disables the batch error processing mode.
**Syntax**

```cpp
virtual void setBatchErrorMode(
    bool batchErrorMode);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>batchErrorMode</td>
<td>TRUE enables batch error processing; FALSE disables batch error processing.</td>
</tr>
</tbody>
</table>

**setBDouble()**

Sets an IEEE754 double as a bind value to a `Statement` object at the position specified by `paramIndex` attribute.

**Syntax**

```cpp
void setBDouble(
    unsigned int paramIndex,
    const BDouble &dval);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>dval</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

**setBfile()**

Sets a parameter to a `Bfile` value.

**Syntax**

```cpp
void setBfile(
    unsigned int paramIndex,
    const Bfile &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

**setBFloat()**

Sets an IEEE754 float as a bind value to a `Statement` object at the position specified by the `paramIndex` attribute.

**Syntax**

```cpp
void setBFloat(
    unsigned int paramIndex,
    const BFloat &fval);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>
setBinaryStreamMode()

Defines that a column is to be returned as a binary stream.

**Syntax**

```cpp
void setBinaryStreamMode(
    unsigned int colIndex,
    unsigned int size);
```

Sets column returned to be a binary stream.

```cpp
void setBinaryStreamMode(
    unsigned int colIndex,
    unsigned int size,
    bool inArg);
```

Sets column returned to be a binary stream; used with PL/SQL IN or IN/OUT arguments in the bind position.

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index; first column is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>size</td>
<td>The amount of data to be read or returned as a binary Stream.</td>
</tr>
<tr>
<td>inArg</td>
<td>Pass TRUE if the bind position is a PL/SQL IN or IN/OUT argument</td>
</tr>
</tbody>
</table>

setBlob()

Sets a parameter to a Blob value.

**Syntax**

```cpp
void setBlob(
    unsigned int paramIndex,
    const Blob &val);`

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

setBytes()

Sets a parameter to a Bytes array.

**Syntax**

```cpp
void setBytes(
    unsigned int paramIndex,
    const Bytes &val);`

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>
setCharacterStreamMode()

Defines that a column is to be returned as a character stream.

Syntax

```c
void setCharacterStreamMode(
    unsigned int colIndex,
    unsigned int size);
```

Sets column returned to be a character stream. Used with PL/SQL IN or IN/OUT arguments in the bind position.

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colIndex</td>
<td>Column index; first column is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>size</td>
<td>The amount of data to be read or returned as a character stream.</td>
</tr>
<tr>
<td>inArg</td>
<td>Pass TRUE if the bind position is a PL/SQL IN or IN/OUT argument</td>
</tr>
</tbody>
</table>

setCharSet()

Overrides the default character set for the specified parameter. Data is assumed to be in the specified character set and is converted to database character set. For OUT binds, this specifies the character set to which database characters are converted to.

Syntax

```c
void setCharSet(
    unsigned int paramIndex,
    string &charSet);
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>charSet</td>
<td>Selected character set, as a string.</td>
</tr>
</tbody>
</table>

setCharSetUString()

Overrides the default character set for the specified parameter. Data is assumed to be in the specified character set and is converted to database character set. For OUT binds, this specifies the character set to which database characters are converted to.

Syntax

```c
void setCharSetUString(
    unsigned int paramIndex,
    const UString& charSet);
```
**setClob()**

Sets a parameter to a Clob value.

**Syntax**

```c
void setClob(
    unsigned int paramIndex,
    const Clob &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>charSet</td>
<td>Selected character set, as a UString.</td>
</tr>
</tbody>
</table>

**setDate()**

Sets a parameter to a Date value.

**Syntax**

```c
void setDate(
    unsigned int paramIndex,
    const Date &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

**setDatabaseNCHARParam()**

If the parameter is going to be inserted in a column that contains data in the database's NCHAR character set, then OCCI must be informed by passing a TRUE value. A FALSE can be passed to restore the default. Returns returns the character set that is in effect for the specified parameter.

**Syntax**

```c
void setDatabaseNCHARParam(
    unsigned int paramIndex,
    bool isNCHAR);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>isNCHAR</td>
<td>TRUE if this parameter contains data in Database's NCHAR character set; FALSE otherwise</td>
</tr>
</tbody>
</table>
setDataBuffer()

Specifies a data buffer where data would be available. Also used for OUT bind parameters of callable statements.

The buffer parameter is a pointer to a user allocated data buffer. The current length of data must be specified in the length parameter. The amount of data should not exceed the size parameter. Finally, type is the data type of the data.

Note that not all types can be supplied in the buffer. For example, all OCCI allocated types (such as Bytes, Date and so on) cannot be provided by the setDataBuffer() interface. Similarly, C++ Standard Library strings cannot be provided with the setDataBuffer() interface either. The type can be any of OCI data types such VARCHAR2, CSTRING, CHAR2 and so on.

If setDataBuffer() is used to specify data for iterative or array executes, it should be called only once in the first iteration only. For subsequent iterations, OCCI would assume that data is at buffer + (i*size) location where i is the iteration number. Similarly the length of the data would be assumed to be at (length+i).

Syntax

```c
void setDataBuffer(
    unsigned int paramIndex,
    void *buffer,
    Type type,
    sb4 size,
    ub2 *length,
    sb2 *ind = NULL,
    ub2 *rc= NULL);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>buffer</td>
<td>Pointer to user-allocated buffer. For iterative or array executes, it should have numIterations() size bytes in it. For array fetches, it should have numRows * size bytes in it. For gather or scatter binds and defines, this structure stores the address of OCIIOVec and the number of OCIIOVec elements that start at that address.</td>
</tr>
<tr>
<td>type</td>
<td>Type of the data that is provided (or retrieved) in the buffer.</td>
</tr>
<tr>
<td>size</td>
<td>Size of the data buffer; for iterative and array executes, it is the size of each element of the data items. For gather or scatter binds and defines, it is the size of the OCIIOVecArray to which the buffer points; the mode must be set to IOVEC.</td>
</tr>
<tr>
<td>length</td>
<td>Pointer to the length of data in the buffer; for iterative and array executes, it should be an array of length data for each buffer element; the size of the array should be equal to arrayLength().</td>
</tr>
<tr>
<td>ind</td>
<td>Indicator. For iterative and array executes, an indicator for every buffer element.</td>
</tr>
<tr>
<td>rc</td>
<td>Returns code; for iterative and array executes, a return code for every buffer element.</td>
</tr>
</tbody>
</table>
setDataBufferArray()

Specifies an array of data buffers where data would be available for reading or writing. Used for IN, OUT, and IN/OUT bind parameters for stored procedures which read/write array parameters.

- A stored procedure can have an array of values for IN, IN/OUT, or OUT parameters. In this case, the parameter must be specified using the setDataBufferArray() method. The array is specified just as for the setDataBuffer() method for iterative or array executes, but the number of elements in the array is determined by *arrayLength parameter.

- For OUT and IN/OUT parameters, the maximum number of elements in the array is specified (and returned) by the arraySize parameter. The client must ensure that it has allocated size *arraySize bytes for the buffer. For iterative prepared statements, the number of elements in the array is determined by the number of iterations, and for array executes the number of elements in the array is determined by the arrayLength parameter of the executeArrayUpdate() method. However, for array parameters of stored procedures, the number of elements in the array must be specified in the arrayLength parameter of the setDataBufferArray() method because each parameter may have a different size array.

- This is different from prepared statements where for iterative and array executes, the number of elements in the array for each parameter is the same and is determined by the number of iterations of the statement, but a callable statement is executed only once, and each of its parameter can be a varying length array with possibly a different length.

Syntax

```c
void setDataBufferArray(
    unsigned int paramIndex,
    void *buffer,
    Type type,
    ub4 arraySize,
    ub4 *arrayLength,
    sb4 elementSize,
    ub2 *elementLength,
    sb2 *ind = NULL,
    ub2 *rc = NULL);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>buffer</td>
<td>Pointer to user-allocated buffer. It should have size* arraySize bytes in it. For gather or scatter binds and defines, this structure stores the address of OCIIOVec and the number of OCIIOVec elements that start at that address.</td>
</tr>
<tr>
<td>type</td>
<td>Type of the data that is provided (or retrieved) in the buffer.</td>
</tr>
<tr>
<td>arraySize</td>
<td>Maximum number of elements in the array.</td>
</tr>
<tr>
<td>arrayLength</td>
<td>Pointer to number of current elements in the array.</td>
</tr>
<tr>
<td>elementSize</td>
<td>Size of the data buffer for each element. For iterative and array executes, it is the size of each element of the data items. When using gather or scatter binds and defines, it is the size of the OCIIOVecArray; the mode must be set to IOVEC.</td>
</tr>
</tbody>
</table>

Oracle C++ Call Interface Programmer's Guide
**setDouble()**

Sets a parameter to a C++ double value.

**Syntax**

```c
void setDouble(
    unsigned int paramIndex,
    double val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

**setErrorOnNull()**

Enables/disables exceptions for reading of NULL values on `paramIndex` parameter of the statement. If exceptions are enabled, calling a `getxxx()` on `paramIndex` parameter would result in an `SQLException` if the parameter value is NULL. This call can also be used to disable exceptions.

**Syntax**

```c
void setErrorOnNull(
    unsigned int paramIndex,
    bool causeException);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>causeException</td>
<td>Enable exceptions if TRUE. Disable if FALSE.</td>
</tr>
</tbody>
</table>

**setErrorOnTruncate()**

This method enables/disables exceptions when truncation occurs.

**Syntax**

```c
void setErrorOnTruncate(
    unsigned int paramIndex,
    bool causeException);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
</tbody>
</table>
setFloat()

Sets a parameter to a C++ float value.

**Syntax**

```c
void setFloat(
    unsigned int paramIndex,
    float val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

setInt()

Sets a parameter to a C++ int value.

**Syntax**

```c
void setInt(
    unsigned int paramIndex,
    int val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

setIntervalDS()

Sets a parameter to a IntervalDS value.

**Syntax**

```c
void setIntervalDS(
    unsigned int paramIndex,
    const IntervalDS &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

setIntervalYM()

Sets a parameter to a Interval value.
Statement Class

Syntax

void setIntervalYM(
    unsigned int paramIndex,
    const IntervalYM &val);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

setMaxIterations()

Sets the maximum number of invocations that are made for the DML statement. This must be called before any parameters are set on the prepared statement. The larger the iterations, the larger the numbers of parameters sent to the server in one round trip. However, a large number causes more memory to be reserved for all the parameters. Note that this is just the maximum limit. Actual number of iterations depends on the number of calls to addIteration().

Syntax

void setMaxIterations(
    unsigned int maxIterations);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxIterations</td>
<td>Maximum number of iterations allowed on this statement.</td>
</tr>
</tbody>
</table>

setMaxParamSize()

This method sets the maximum amount of data to be sent or received for the specified parameter. It only applies to character and binary data. If the maximum amount is exceeded, the excess data is discarded. This method can be very useful when working with a LONG column. It can be used to truncate the LONG column by reading or writing it into a string or Bytes data type.

If the getSQL() or setBytes() method has been called to bind a value to an IN/OUT parameter of a PL/SQL procedure, and the size of the OUT value is expected to be greater than the size of the IN value, then setMaxParamSize() should be called.

Syntax

void setMaxParamSize(
    unsigned int paramIndex,
    unsigned int maxSize);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>maxSize</td>
<td>The new maximum parameter size limit; must be &gt;0.</td>
</tr>
</tbody>
</table>
**setNull()**

Sets a parameter to SQL NULL. Note that you must specify the parameter's SQL type.

### Syntax

```c++
void setNull(
    unsigned int paramIndex,
    Type type);
```

Sets the value of a parameter to NULL using an object.

```c++
void setNull(
    unsigned int paramIndex,
    Type type,
    const string &typeName,
    const string &schemaName = "")
```

Sets the value of a parameter to NULL for object and collection types, `OCCIPOBJ`ECT and `OCCIVECTOR`. Uses the appropriate schema and type name of the object or collection type. Support for string.

```c++
void setNull(
    unsigned int paramIndex,
    Type type,
    UString &typeName,
    UString &schemaName);
```

Sets the value of a parameter to NULL for object and collection types, `OCCIPOBJ`ECT and `OCCIVECTOR`. Uses the appropriate schema and type name of the object or collection type. Support for UString.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>type</td>
<td>SQL type.</td>
</tr>
<tr>
<td>typeName</td>
<td>Type name of the object or collection.</td>
</tr>
<tr>
<td>schemaName</td>
<td>Name of the schema where the object or collection is defined.</td>
</tr>
</tbody>
</table>

**setNumber()**

Sets a parameter to a Number value.

### Syntax

```c++
void setNumber(
    unsigned int paramIndex,
    const Number &val);
```

```
```

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

**setObject()**

Sets the value of a parameter using an object; use the C++.lang equivalent objects for integral values. The OCCI specification specifies a standard mapping from C++ Object types to SQL types. The given parameter C++ object is converted to the corresponding SQL type before being sent to the database.
Syntax

```c
void setObject(
    unsigned int paramIndex,
    PObject* val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The object containing the input parameter value.</td>
</tr>
<tr>
<td>sqltyp</td>
<td>The SQL type name of the object to be set.</td>
</tr>
</tbody>
</table>

**setPrefetchMemorySize()**

Sets the amount of memory that is used internally by OCCI to store data fetched during each round trip to the server. A value of 0 means that the amount of data fetched during the round trip is constrained by the `FetchRowCount` parameter. If both parameters are nonzero, the smaller of the two is used.

Syntax

```c
void setPrefetchMemorySize(
    unsigned int bytes);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>Number of bytes used for storing data fetched during each server round trip.</td>
</tr>
</tbody>
</table>

**setPrefetchRowCount()**

Sets the number of rows that are fetched internally by OCCI during each round trip to the server. A value of 0 means that the amount of data fetched during the round trip is constrained by the `FetchMemorySize` parameter. If both parameters are nonzero, the smaller of the two is used. If both of these parameters are zero, row count internally defaults to 1 row and that is the value returned from the `getFetchRowCount()` method.

Syntax

```c
void setPrefetchRowCount(
    unsigned int rowCount);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowCount</td>
<td>Number of rows to fetch for each round trip to the server.</td>
</tr>
</tbody>
</table>

**setRef()**

Sets the value of a parameter to a reference. A `Ref<T>` instance is implicitly converted to a `RefAny` object during this call.
**setRowid()**

Sets a `Rowid` bytes array for a bind position.

**Syntax**

```c++
void setRowid(
    unsigned int paramIndex,
    const Bytes &val);
```

**Description**

Sets the value of a parameter to a reference.

**Parameter Description**

- `paramIndex`: Parameter index; first parameter is 1, second is 2, and so on.
- `val`: The parameter value.

---

**setSQL()**

A new SQL string can be associated with a `Statement` object using this call. Resources associated with the previous SQL statement are freed. In particular, a previously obtained result set is invalidated. If an empty SQL string, """, is used when the `Statement` is created, a `setSQL` method with the proper SQL string must be performed first.

**Syntax**

```c++
void setSQL(
    const string &sql);
```

**Parameter Description**

- `sql`: The SQL string.

---

**setRef()**

Sets the value of a parameter to a reference.

**Syntax**

```c++
void setRef(
    unsigned int paramIndex,
    const RefAny &refAny);
```

**Sets the value of a parameter to a reference.**

**Parameter Description**

- `paramIndex`: Parameter index; first parameter is 1, second is 2, and so on.
- `refAny`: The reference.

---

**Syntax**

```c++
void setRef(
    unsigned int paramIndex,
    const RefAny &refAny,
    const string &typName,
    const string &schName = "");
```

**Sets the value of a parameter to a reference. If the `Statement` represents a callable PL/SQL method, pass the schema name and type name of the object represented by the `Ref`. Support for `string`.

**Parameter Description**

- `paramIndex`: Parameter index; first parameter is 1, second is 2, and so on.
- `refAny`: The reference.
- `typName`: The type of the object [optional].
- `schName`: The schema where the object type is defined [optional].

---

**Syntax**

```c++
void setRef(
    unsigned int paramIndex,
    const RefAny &refAny,
    const UString &typName,
    const UString &schName);
```

**Sets the value of a parameter to a reference. If the `Statement` represents a callable PL/SQL method, pass the schema name and type name of the object represented by the `Ref`. Support for `UString`.

**Parameter Description**

- `paramIndex`: Parameter index; first parameter is 1, second is 2, and so on.
- `refAny`: The reference.
- `typName`: The type of the object [optional].
- `schName`: The schema where the object type is defined [optional].
**setSQLUString()**

Associate an SQL statement with this object. Unicode support: the client Environment should be initialized in OCCIUTIF16 mode.

**Syntax**

```cpp
void setSQLUString(
    const UString &sql);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td>A SQL statement in same character set as the connection source of the statement.</td>
</tr>
</tbody>
</table>

**setString()**

Sets a parameter for a specified index.

**Syntax**

```cpp
void setString(
    unsigned int paramIndex,
    const string &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

**setTimestamp()**

Sets a parameter to a `Timestamp` value.

**Syntax**

```cpp
void setTimestamp(
    unsigned int paramIndex,
    const Timestamp &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

**setUInt()**

Sets a parameter to a C++ `unsigned int` value.
Syntax

```c
void setUInt(
    unsigned int paramIndex,
    unsigned int val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

**setUString()**

Sets a parameter for a specified index; globalization enabled.

**Syntax**

```c
void setUString(
    unsigned int paramIndex,
    const UString &val);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramIndex</td>
<td>Parameter index; first parameter is 1, second is 2, and so on.</td>
</tr>
<tr>
<td>val</td>
<td>The parameter value.</td>
</tr>
</tbody>
</table>

**setVector()**

Sets a parameter to a vector. This method is necessary when the type is a collection type, varrays or nested tables. The SQL Type of the elements in the collection should be compatible with the type of the vector. For example, if the collection is a varray of VARCHAR2, use `vector<string>`.

**Syntax**

```c
void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<T> &vect,
    const string &schemaName,
    const string &typeName);
```

Sets a `BDouble` vector.

- Intended for use on platforms where partial ordering of function templates is supported. Multibyte support.
- Intended for use on platforms where partial ordering of function templates is not supported, such as Windows NT and z/OS.
- Multibyte support.
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`void setVector(</td>
<td>Sets a const Bfile vector;</td>
</tr>
<tr>
<td>Statement *stmt,</td>
<td>multibyte support.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>const vector&lt;Bfile&gt; &amp;vect,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;schemaName,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;typeName);</td>
<td></td>
</tr>
<tr>
<td>);</td>
<td></td>
</tr>
<tr>
<td>`void setVector(</td>
<td>Sets a const BFILE vector;</td>
</tr>
<tr>
<td>Statement *stmt,</td>
<td>UTF16 support.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>const vector&lt;Bfile&gt; &amp;vect,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;schemaName,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;typeName);</td>
<td></td>
</tr>
<tr>
<td>);</td>
<td></td>
</tr>
<tr>
<td>`void setVector(</td>
<td>Sets a BFloat vector.</td>
</tr>
<tr>
<td>Statement *stmt,</td>
<td></td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>const vector&lt;BFloat&gt; &amp;vect,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;sqltype);</td>
<td></td>
</tr>
<tr>
<td>);</td>
<td></td>
</tr>
<tr>
<td>`void setVector(</td>
<td>Sets a const Blob vector;</td>
</tr>
<tr>
<td>Statement *stmt,</td>
<td>UTF16 support.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>const vector&lt;Blob&gt; &amp;vect,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;schemaName,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;typeName);</td>
<td></td>
</tr>
<tr>
<td>);</td>
<td></td>
</tr>
<tr>
<td>`void setVector(</td>
<td>Sets a const Blob vector;</td>
</tr>
<tr>
<td>Statement *stmt,</td>
<td>UTF16 support.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>const vector&lt;Blob&gt; &amp;vect,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;schemaName,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;typeName);</td>
<td></td>
</tr>
<tr>
<td>);</td>
<td></td>
</tr>
<tr>
<td>`void setVector(</td>
<td>Sets a const Clob vector;</td>
</tr>
<tr>
<td>Statement *stmt,</td>
<td>UTF16 support.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>const vector&lt;Clob&gt; &amp;vect,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;schemaName,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;typeName);</td>
<td></td>
</tr>
<tr>
<td>);</td>
<td></td>
</tr>
<tr>
<td>`void setVector(</td>
<td>Sets a const Clob vector;</td>
</tr>
<tr>
<td>Statement *stmt,</td>
<td>UTF16 support.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>const vector&lt;Clob&gt; &amp;vect,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;schemaName,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;typeName);</td>
<td></td>
</tr>
<tr>
<td>);</td>
<td></td>
</tr>
<tr>
<td>`void setVector(</td>
<td>Sets a const Date vector;</td>
</tr>
<tr>
<td>Statement *stmt,</td>
<td>UTF16 support.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>const vector&lt;Date&gt; &amp;vect,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;schemaName,</td>
<td></td>
</tr>
<tr>
<td>const string &amp;typeName);</td>
<td></td>
</tr>
<tr>
<td>);</td>
<td></td>
</tr>
<tr>
<td>`void setVector(</td>
<td>Sets a const Date vector;</td>
</tr>
<tr>
<td>Statement *stmt,</td>
<td>UTF16 support.</td>
</tr>
<tr>
<td>unsigned int paramIndex,</td>
<td></td>
</tr>
<tr>
<td>const vector&lt;Date&gt; &amp;vect,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;schemaName,</td>
<td></td>
</tr>
<tr>
<td>const UString &amp;typeName);</td>
<td></td>
</tr>
<tr>
<td>);</td>
<td></td>
</tr>
<tr>
<td>Syntax</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
</tbody>
</table>
| `void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<double> &vect,
  const string &schemaName,
  const string &typeName);
`                          | Sets a const double vector;       |
|                                                                      | multibyte support.               |
| `void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<double> &vect,
  const UString &schemaName,
  const UString &typeName);
`                          | Sets a const double vector;       |
|                                                                      | UTF16 support.                   |
| `void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<float> &vect,
  const string &schemaName,
  const string &typeName);
`                          | Sets a const float vector;        |
|                                                                      | multibyte support.               |
| `void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<float> &vect,
  const UString &schemaName,
  const UString &typeName);
`                          | Sets a const float vector;        |
|                                                                      | UTF16 support.                   |
| `void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<int> &vect,
  const string &schemaName,
  const string &typeName);
`                          | Sets a const int vector;          |
|                                                                      | multibyte support.               |
| `void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<int> &vect,
  const UString &schemaName,
  const UString &typeName);
`                          | Sets a const int vector;          |
|                                                                      | UTF16 support.                   |
| `void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<IntervalDS> &vect,
  const string &schemaName,
  const string &typeName);
`                          | Sets a const IntervalDS vector;   |
|                                                                      | multibyte support.               |
| `void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<IntervalDS> &vect,
  const UString &schemaName,
  const UString &typeName);
`                          | Sets a const IntervalDS vector;   |
|                                                                      | UTF16 support.                   |
| `void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<IntervalYM> &vect,
  const string &schemaName,
  const string &typeName);
`                          | Sets a const IntervalYM vector;   |
|                                                                      | multibyte support.               |
| `void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<IntervalYM> &vect,
  const UString &schemaName,
  const UString &typeName);
`                          | Sets a const IntervalYM vector;   |
void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<IntervalYM> &vect,
    const UString &schemaName,
    const UString &typeName);

Sets a const IntervalYM vector; UTF16 support

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<Number> &vect,
    const string &schemaName,
    const string &typeName);

Sets a const Number vector; multibyte support.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<Number> &vect,
    const UString &schemaName,
    const UString &typeName);

Sets a const Number vector; UTF16 support.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<RefAny> &vect,
    const string &schemaName,
    const string &typeName);

Sets a const RefAny vector; multibyte support.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<RefAny> &vect,
    const UString &schemaName,
    const UString &typeName);

Sets a const RefAny vector; UTF16 support.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<string> &vect,
    const string &schemaName,
    const string &typeName);

Sets a const string vector; multibyte support.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<string> &vect,
    const UString &schemaName,
    const UString &typeName);

Sets a const string vector; UTF16 support.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<Timestamp> &vect,
    const string &schemaName,
    const string &typeName);

Sets a const Timestamp vector; multibyte support.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<Timestamp> &vect,
    const UString &schemaName,
    const UString &typeName);

Sets a const Timestamp vector; UTF16 support.
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
</table>
| void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<unsigned int> &vect,
  const string &schemaName,
  const string &typeName); | Sets a const unsigned int vector; multibyte support. |
| void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  const vector<unsigned int> &vect,
  const UString &schemaName,
  const UString &typeName); | Sets a const unsigned int vector; UTF16 support. |
| void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  vector<Bfile> &vect,
  string &sqltype); | Sets a Bfile vector. |
| void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  vector<Blob> &vect,
  string &sqltype); | Sets a Blob vector. |
| void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  vector<Clob> &vect,
  string &sqltype); | Sets a Clob vector. |
| void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  vector<Date> &vect,
  string &sqltype); | Sets a Date vector. |
| void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  vector<double> &vect,
  string &sqltype); | Sets a double vector. |
| void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  vector<float> &vect,
  string &sqltype); | Sets a float vector. |
| void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  vector<int> &vect,
  string &sqltype); | Sets an int vector. |
| void setVector(
  Statement *stmt,
  unsigned int paramIndex,
  vector<IntervalDS> &vect,
  string &sqltype); | Sets an IntervalDS vector. |
void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    vector<IntervalYM> &vect,
    string &sqltype);

Sets an IntervalYM vector.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    vector<Number> &vect,
    string &sqltype);

Sets a Number vector.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    vector<RefAny> &vect,
    string &sqltype);

Sets a RefAny vector.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    vector<string> &vect,
    string &sqltype);

Sets a string vector.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    vector<Timestamp> &vect,
    string &sqltype);

Sets a Timestamp vector.

void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    vector<unsigned int> &vect,
    string &sqltype);

Sets an unsigned int vector.

template <class T>
void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<T*> &vect,
    const string &sqltype);

Intended for use on platforms where partial ordering of function templates is not supported.

template <class T>
void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    const vector<T> &vect,
    const string &sqltype);

Intended for use on platforms where partial ordering of function templates is supported.

template <class T>
void setVector(
    Statement *stmt,
    unsigned int paramIndex,
    vector<Ref<T>> &vect,
    string &sqltype);

Available only on platforms where partial ordering of function templates is supported. setVectorOfRefs() can be used instead.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmt</td>
<td>Statement on which parameter is to be set.</td>
</tr>
</tbody>
</table>
setVectorOfRefs()

Sets a parameter to a vector; is necessary when the type is a collection of REFS or nested tables of REFS.

template <class T> void setVectorOfRefs(  
    Statement *stmt, unsigned int paramIndex,  
    const vector<Ref<T> > &vect,  
    const string &sqltype);  
Sets a parameter to a vector; is necessary when the type is a collection of REFS are varrays or nested tables of REFS.

template <class T> void setVectorOfRefs(  
    Statement *stmt,  
    unsigned int paramIndex,  
    const vector<Ref<T> > &vect,  
    const string &schemaName,  
    const string &typeName);  
Used for multibyte support.

template <class T> void setVectorOfRefs(  
    Statement *stmt,  
    unsigned int paramIndex,  
    const vector<Ref<T> > &vect,  
    const UString &schemaName,  
    const UString &typeName);  
Used for UTF16 support on platforms where partial ordering of function templates is supported.

Parameter | Description
---|---
stmt | Statement on which parameter is to be set.
paramIndex | Parameter index; first parameter is 1, second is 2, and so on.
vect | Vector to be set.
status()

Returns the current status of the statement. Useful when there is streamed data to be written (or read). Other methods such as `getCurrentStreamParam()` and `getgetCurrentIteration()` can be called to find out the streamed parameter that must be written and the current iteration number for an iterative or array execute. The `status()` method can be called repeatedly to find out the status of the execution.

The returned value, Status, is defined in Table 13–43 on page 13-213.

**Syntax**

```cpp
Status status() const;
```
Stream Class

You use a Stream to read or write streamed data (usually LONG).

- A read-able Stream is used to obtain streamed data from a result set or OUT bind variable from a stored procedure call. A read-able Stream must be read completely until the end of data is reached or it should be closed to discard any unwanted data.

- A write-able Stream is used to provide streamed data (usually LONG) to parameterized statements including callable statements.

### Table 13–45 Enumerated Values Used by Stream Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>READY_FOR_READ indicates that the Stream is ready for read operations</td>
</tr>
<tr>
<td></td>
<td>READY_FOR_WRITE indicates that the Stream is ready for write operations</td>
</tr>
<tr>
<td></td>
<td>INACTIVE indicates that the Stream is not available for ready or write operations</td>
</tr>
</tbody>
</table>

### Table 13–46 Summary of Stream Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>readBuffer() on page 13-254</td>
<td>Reads the stream and returns the amount of data read from the Stream object.</td>
</tr>
<tr>
<td>readLastBuffer() on page 13-255</td>
<td>Reads last buffer from Stream.</td>
</tr>
<tr>
<td>writeBuffer() on page 13-255</td>
<td>Writes data from buffer to the stream.</td>
</tr>
<tr>
<td>writeLastBuffer() on page 13-255</td>
<td>Writes the last data from buffer to the stream.</td>
</tr>
<tr>
<td>status() on page 13-255</td>
<td>Returns the current status of the stream.</td>
</tr>
</tbody>
</table>

### readBuffer()

Reads data from Stream. The size parameter specifies the maximum number of byte characters to read. Returns the amount of data read from the Stream object. -1 means end of data on the stream.

### Syntax

```c++
virtual int readBuffer(
    char *buffer,
    unsigned int size) = 0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>Pointer to data buffer; must be allocated and freed by caller.</td>
</tr>
<tr>
<td>size</td>
<td>Specifies the number of bytes to be read.</td>
</tr>
</tbody>
</table>
readLastBuffer()

Reads the last buffer from the Stream. It can also be called to discard unread data. The size parameter specifies the maximum number of byte characters to read. Returns the amount of data read from the Stream object; -1 means end of data on the stream.

Syntax

```cpp
virtual int readLastBuffer(
    char *buffer,
    unsigned int size) = 0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>Pointer to data buffer; must be allocated and freed by caller.</td>
</tr>
<tr>
<td>size</td>
<td>Specifies the number of bytes to be read.</td>
</tr>
</tbody>
</table>

writeBuffer()

Writes data from buffer to the stream. The amount of data is determined by size.

Syntax

```cpp
virtual void writeBuffer(
    char *buffer,
    unsigned int size) = 0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>Pointer to data buffer.</td>
</tr>
<tr>
<td>size</td>
<td>Specifies the number of chars to be written.</td>
</tr>
</tbody>
</table>

writeLastBuffer()

This method writes the last data buffer to the stream. It can also be called to write the last chunk of data. The amount of data written is determined by size.

Syntax

```cpp
virtual void writeLastBuffer(
    char *buffer,
    unsigned int size) = 0;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>Pointer to data buffer.</td>
</tr>
<tr>
<td>size</td>
<td>Specifies the number of bytes to be written.</td>
</tr>
</tbody>
</table>

status()

Returns the current Status, as defined in Table 13-45 on page 13-254.
Syntax

virtual Status status() const;
Subscription Class

The subscription class encapsulates the information and operations necessary for registering a subscriber for notification.

*Table 13–47  Enumerated Values Used by Subscription Class*

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presentation</td>
<td>■ PRES_DEFAULT indicates that the event notification should be in</td>
</tr>
<tr>
<td></td>
<td>default format.</td>
</tr>
<tr>
<td></td>
<td>■ PRES_XML indicates that the event notification should be in XML</td>
</tr>
<tr>
<td></td>
<td>format.</td>
</tr>
<tr>
<td>Protocol</td>
<td>■ PROTO_CBK indicates that the client receives notifications through</td>
</tr>
<tr>
<td></td>
<td>the default system protocol.</td>
</tr>
<tr>
<td></td>
<td>■ PROTO_MAIL indicates that the client receives notifications through</td>
</tr>
<tr>
<td></td>
<td>e-mail, like <a href="mailto:xyz@oracle.com">xyz@oracle.com</a>. The database does not check if the</td>
</tr>
<tr>
<td></td>
<td>e-mail is valid.</td>
</tr>
<tr>
<td></td>
<td>■ PROTO_SERVER indicates that the client receives notifications</td>
</tr>
<tr>
<td></td>
<td>through an invoked PL/SQL procedure in the database, like</td>
</tr>
<tr>
<td></td>
<td>schema.procedure. The subscriber must have the appropriate</td>
</tr>
<tr>
<td></td>
<td>permissions on the procedure.</td>
</tr>
<tr>
<td></td>
<td>■ PROTO_HTTP indicates that the client receives notifications through</td>
</tr>
<tr>
<td></td>
<td>an HTTP URL, like <a href="http://www.oracle.com:80">http://www.oracle.com:80</a>. The database does not</td>
</tr>
<tr>
<td></td>
<td>check if the URL is valid.</td>
</tr>
<tr>
<td>Namespace</td>
<td>■ NS_ANONYMOUS indicates that the registrations are made in an</td>
</tr>
<tr>
<td></td>
<td>anonymous namespace.</td>
</tr>
<tr>
<td></td>
<td>■ NS_AQ indicates that the registrations are made in the Oracle</td>
</tr>
<tr>
<td></td>
<td>Streams Advanced Queuing namespace.</td>
</tr>
</tbody>
</table>

*Table 13–48  Summary of Subscription Methods*

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscription() on page 13-258</td>
<td>Subscription class constructor.</td>
</tr>
<tr>
<td>getCallbackContext() on page 13-259</td>
<td>Retrieves the callback context.</td>
</tr>
<tr>
<td>getDatabaseServersCount() on page 13-259</td>
<td>Retrieves the number of database servers in which the client is interested for the registration.</td>
</tr>
<tr>
<td>getDatabaseServerNames() on page 13-259</td>
<td>Returns the names of all the database servers where the client registered an interest for notification.</td>
</tr>
<tr>
<td>getNotifyCallback() on page 13-259</td>
<td>Returns the pointer to the registered callback function.</td>
</tr>
<tr>
<td>getPayload() on page 13-259</td>
<td>Retrieves the payload that has been set on the Subscription object before posting.</td>
</tr>
<tr>
<td>getSubscriptionName() on page 13-260</td>
<td>Retrieves the name of the Subscription.</td>
</tr>
<tr>
<td>getSubscriptionNamespace() on page 13-260</td>
<td>Retrieves the namespace of the Subscription.</td>
</tr>
<tr>
<td>getRecipientName() on page 13-260</td>
<td>Retrieves the name of the Subscription recipient.</td>
</tr>
<tr>
<td>getPresentation() on page 13-260</td>
<td>Retrieves the notification presentation mode.</td>
</tr>
</tbody>
</table>
Subscription() class constructor.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscription (</td>
<td>Creates a Subscription within a specified</td>
</tr>
<tr>
<td>const Environment *env</td>
<td>Environment.</td>
</tr>
<tr>
<td>);</td>
<td></td>
</tr>
<tr>
<td>Subscription(const</td>
<td>Copy constructor.</td>
</tr>
<tr>
<td>Subscription&amp; sub);</td>
<td></td>
</tr>
</tbody>
</table>

Syntax

Subscription(const Subscription& sub);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>The Environment.</td>
</tr>
</tbody>
</table>
getCallbackContext()

Retrieves the callback context.

**Syntax**

```cpp
void* getCallbackContext() const;
```

getDatabaseServersCount()

Returns the number of database servers in which the client is interested for the registration.

**Syntax**

```cpp
unsigned int getDatabaseServersCount() const;
```

getDatabaseServerNames()

Returns the names of all the database servers where the client registered an interest for notification.

**Syntax**

```cpp
vector<string> getDatabaseServerNames() const;
```

getNotifyCallback()

Returns the pointer to the callback function registered for this `Subscription`.

**Syntax**

```cpp
unsigned int (*getNotifyCallback() const)(
    Subscription& sub,
    NotifyResult *nr);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>The original <code>Subscription</code>.</td>
</tr>
<tr>
<td>nr</td>
<td>The <code>NotifyResult</code>.</td>
</tr>
</tbody>
</table>

getPayload()

Retrieves the payload that has been set on the `Subscription` object before posting.

**Syntax**

```cpp
Bytes getCPayload() const;
```
getSubscriptionName()

Retrieves the name of the subscription.

**Syntax**
string getSubscriptionName() const;

getSubscriptionNamespace()

Retrieves the namespace of the subscription. The subscription name must be consistent
with its namespace. Valid *Namespace* values are NS_AQ and NS_ANONYMOUS, as
defined in Table 13–47 on page 13-257.

**Syntax**
Namespace getSubscriptionNamespace() const;

getRecipientName()

Retrieves the name of the recipient of the notification. Possible return values are E-mail
address, the HTTP url and the PL/SQL procedure, depending on the protocol.

**Syntax**
string getRecipientName() const;

getPresentation()

Retrieves the presentation mode in which the client receives notifications. Valid
*Presentation* values are defined in Table 13–47 on page 13-257.

**Syntax**
Presentation getPresentation() const;

getProtocol()

Retrieves the protocol in which the client receives notifications. Valid *Protocol*
values are defined in Table 13–47 on page 13-257.

**Syntax**
Protocol getProtocol() const;

isNull()

Returns **TRUE** if *Subscription* is **NULL** or **FALSE** otherwise.

**Syntax**
bool isNull() const;
operator=()

Assignment operator for Subscription.

**Syntax**

```c
void operator=(
    const Subscription& sub);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sub</code></td>
<td>The original Subscription.</td>
</tr>
</tbody>
</table>

**setCallbackContext()**

Registers a notification callback function when the protocol is PROTO_CBK, as defined in Table 13–47 on page 13-257. Context registration is also included in this call.

**Syntax**

```c
void setCallbackContext(
    void *ctx);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ctx</code></td>
<td>The context set.</td>
</tr>
</tbody>
</table>

**setDatabaseServerNames()**

Specifies the list of database server distinguished names from which the client receives notifications.

**Syntax**

```c
void setDatabaseServerNames(
    const vector<string>& dbsrv);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbsrv</code></td>
<td>The list of database distinguished names</td>
</tr>
</tbody>
</table>

**setNotifyCallback()**

Sets the context that the client wants to get passed to the user callback. If the protocol is set to PROTO_CBK or not specified, this attribute must be set before registering the subscription handle.

**Syntax**

```c
void setNotifyCallback(
    unsigned int (*callback)(
        Subscription& sub,
        void *ctx),
    void *ctx);
```
setNull()

Sets the Subscription object to NULL and frees the memory associated with the object.

**Syntax**

```cpp
void setNull();
```

setPayload()

Sets the buffer content that corresponds to the payload to be posted to the Subscription.

**Syntax**

```cpp
void setPayload(
    const Bytes& payload);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>payload</td>
<td>Content of the notification.</td>
</tr>
</tbody>
</table>

setPresentation()

Sets the presentation mode in which the client receives notifications.

**Syntax**

```cpp
void setPresentation(
    Presentation pres);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pres</td>
<td>Presentation mode, as defined in Table 13–47 on page 13-257.</td>
</tr>
</tbody>
</table>

setProtocol()

Sets the Protocol in which the client receives event notifications, as defined in Table 13–47 on page 13-257.

**Syntax**

```cpp
void setProtocol(
    Protocol prot);
```
**setSubscriptionName()**

Sets the name of the subscription. All subscriptions are identified by a subscription name, which consists of a sequence of bytes of specified length.

If the namespace is `NS_AQ`, the subscription name is:
- `SCHEMA.QUEUE` when registering on a single consumer queue
- `SCHEMA.QUEUE:CONSUMER_NAME` when registering on a multiconsumer queue

**Syntax**

```cpp
void setSubscriptionName(
    const string& name);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Subscription name.</td>
</tr>
</tbody>
</table>

**setSubscriptionNamespace()**

Sets the namespace where the subscription is used. The subscription name must be consistent with its namespace. Default value is `NS_AQ`.

**Syntax**

```cpp
void setSubscriptionNamespace(
    Namespace nameSpace);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nameSpace</td>
<td>Namespace in which the subscription is used, as defined in Table 13–47 on page 13-257.</td>
</tr>
</tbody>
</table>

**setRecipientName()**

Sets the name of the recipient of the notification.

**Syntax**

```cpp
void setRecipientName(
    const string& name);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the notification recipient.</td>
</tr>
</tbody>
</table>
This class supports the SQL standard `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITHOUT TIME ZONE` types, and works with all database `TIMESTAMP` types: `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `TIMESTAMP WITH LOCAL TIME ZONE`.

`Timestamp` time components, such as hour, minute, second and fractional section are in the time zone specified for the `Timestamp`. This is new behavior for the 10g release; previous versions supported GMT values of time components. Time components were only converted to the time zone specified by `Timestamp` when they were stored in the database. For example, the following `Timestamp()` call constructs a `Timestamp` value `13-Nov 2003 17:24:30.0` in timezone `+5:30`.

```
Timestamp ts(env, 2003, 11, 13, 17, 24, 30, 0, 5, 30);
```

The behavior of this call in previous releases would interpret the timestamp components as GMT, resulting in a timestamp value of `13-Nov 2003 11:54:30.0` in timezone `+5:30`. Users were forced to convert the timestamps to GMT before invoking the constructor. Note that for GMT timezone, both hour and minute equal 0.

This behavior change also applies to `setDate()` and `setTime()` methods.

The fields of `Timestamp` class and their legal ranges are provided in Table 13–49. An `SQLException` occurs if a parameter is out of range.

**Table 13–49** Fields of Timestamp and Their Legal Ranges

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Minimum Value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>int</td>
<td>-4713</td>
<td>9999</td>
</tr>
<tr>
<td>month</td>
<td>unsigned int</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>day</td>
<td>unsigned int</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>hour</td>
<td>unsigned int</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>min</td>
<td>unsigned int</td>
<td>0</td>
<td>59</td>
</tr>
<tr>
<td>sec</td>
<td>unsigned int</td>
<td>0</td>
<td>61</td>
</tr>
<tr>
<td>tzhour</td>
<td>int</td>
<td>-12</td>
<td>14</td>
</tr>
<tr>
<td>tzmin</td>
<td>int</td>
<td>-59</td>
<td>59</td>
</tr>
</tbody>
</table>

**Table 13–50** Summary of Timestamp Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Timestamp()</code> on page 13-265</td>
<td><code>Timestamp</code> class constructor.</td>
</tr>
<tr>
<td><code>fromText()</code> on page 13-267</td>
<td>Sets the time stamp from the values provided by the string.</td>
</tr>
<tr>
<td><code>getDate()</code> on page 13-268</td>
<td>Gets the date from the <code>Timestamp</code> object.</td>
</tr>
<tr>
<td><code>getTime()</code> on page 13-268</td>
<td>Gets the time from the <code>Timestamp</code> object.</td>
</tr>
<tr>
<td><code>getTimeZoneOffset()</code> on page 13-269</td>
<td>Returns the time zone hour and minute offset value.</td>
</tr>
<tr>
<td><code>intervalAdd()</code> on page 13-269</td>
<td>Returns a <code>Timestamp</code> object with value (this + interval).</td>
</tr>
</tbody>
</table>
Table 13–50  (Cont.) Summary of Timestamp Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>intervalSub() on page 13-269</td>
<td>Returns a Timestamp object with value (this - interval).</td>
</tr>
<tr>
<td>isNull() on page 13-270</td>
<td>Checks if Timestamp is NULL.</td>
</tr>
<tr>
<td>operator=() on page 13-270</td>
<td>Simple assignment.</td>
</tr>
<tr>
<td>operator==() on page 13-270</td>
<td>Checks if a and b are equal.</td>
</tr>
<tr>
<td>operator!=() on page 13-270</td>
<td>Checks if a and b are not equal.</td>
</tr>
<tr>
<td>operator&gt;() on page 13-271</td>
<td>Checks if a is greater than b.</td>
</tr>
<tr>
<td>operator&gt;=() on page 13-271</td>
<td>Checks if a is greater than or equal to b.</td>
</tr>
<tr>
<td>operator&lt;() on page 13-271</td>
<td>Checks if a is less than b.</td>
</tr>
<tr>
<td>operator&lt;=() on page 13-272</td>
<td>Checks if a is less than or equal to b.</td>
</tr>
<tr>
<td>setDate() on page 13-272</td>
<td>Sets the year, month, day components contained for this timestamp.</td>
</tr>
<tr>
<td>setNull() on page 13-272</td>
<td>Sets the value of Timestamp to NULL</td>
</tr>
<tr>
<td>setTime() on page 13-272</td>
<td>Sets the day, hour, minute, second and fractional second components for this timestamp.</td>
</tr>
<tr>
<td>setTimeZoneOffset() on page 13-273</td>
<td>Sets the hour and minute offset for time zone.</td>
</tr>
<tr>
<td>subDS() on page 13-273</td>
<td>Returns an IntervalDS representing this - val.</td>
</tr>
<tr>
<td>subYM() on page 13-273</td>
<td>Returns an IntervalYM representing this - val.</td>
</tr>
<tr>
<td>toText() on page 13-274</td>
<td>Returns a string representation for the timestamp in the format specified.</td>
</tr>
</tbody>
</table>

Timestamp()

Timestamp class constructor.

Syntax Description

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamp{</td>
<td>Returns a default Timestamp object. Time components are understood to be in the specified time zone.</td>
</tr>
<tr>
<td>const Environment *env,</td>
<td></td>
</tr>
<tr>
<td>int year=1,</td>
<td></td>
</tr>
<tr>
<td>unsigned int month=1,</td>
<td></td>
</tr>
<tr>
<td>unsigned int day=1,</td>
<td></td>
</tr>
<tr>
<td>unsigned int hour=0,</td>
<td></td>
</tr>
<tr>
<td>unsigned int min=0,</td>
<td></td>
</tr>
<tr>
<td>unsigned int sec=0,</td>
<td></td>
</tr>
<tr>
<td>unsigned int fs=0,</td>
<td></td>
</tr>
<tr>
<td>int tzhour=0,</td>
<td></td>
</tr>
<tr>
<td>int tzmin=0};</td>
<td></td>
</tr>
<tr>
<td>Timestamp();</td>
<td>Returns a NULL Timestamp object. A NULL timestamp can be initialized by assignment or calling the fromText() method. Methods that can be called on NULL timestamp objects are setNull(), isNull() and operator=.</td>
</tr>
</tbody>
</table>
Example 13–11 Using Default Timestamp Constructor

This example demonstrates that the default constructor creates a NULL value, and how you can assign a non-NULL value to a Timestamp and perform operations on it:

```cpp
Environment *env = Environment::createEnvironment();
//create a null timestamp
Timestamp ts;
if(ts.isNull())
    cout << "\n ts is Null";

//assign a non null value to ts
Timestamp notNullTs(env, 2000, 8, 17, 12, 0, 0, 0, 5, 30);
```
ts = notNullTs;

//now all operations are valid on ts
int yr;
unsigned int mth, day;
ts.setDate(yr, mth, day);

**Example 13–12  Using fromText() method to Initialize a NULL Timestamp Instance**
The following code example demonstrates how to use the fromText() method to initialize a NULL timestamp:

Environment *env = Environment::createEnvironment();

Timestamp tsl;
 tsl.fromText("01:16:17.12 04/03/1825", "hh:mi:ss ff dd/mm/yyyy", "", env);

**Example 13–13  Comparing Timestamps Stored in the Database**
The following code example demonstrates how to get the timestamp column from a result set, check whether the timestamp is NULL, get the timestamp value in string format, and determine the difference between 2 timestamps:

Timestamp reft(env, 2001, 1, 1);
ResultSet *rs=stmt->executeQuery(" select order_date from orders where customer_id=1");
rs->next();

//retrieve the timestamp column from result set
Timestamp ts=rs->getTimestamp(1);

//check timestamp for null
if(!ts.isNull())
{
    string tsstr=ts.toText(); //get the timestamp value in string format
    "dd/mm/yyyy hh:mm:ss [tzh:tzm]",0);
    if(reft<ts //compare timestamps
        IntervalDS ds=reft.subDS(ts); //get difference between timestamps
}

fromText()

Sets the timestamp value from the string. The string is expected to be in the format specified. If nlsParam is specified, this determines the NLS parameters to be used for the conversion. If nlsParam is not specified, the NLS parameters are picked up from the environment which has been passed. In case environment is not passed, Globalization Support parameters are obtained from the environment associated with the instance, if any.

Sets Timestamp object to value represented by a string or UString.

The value is interpreted based on the fmt and nlsParam parameters. In cases where nlsParam is not passed, the Globalization Support settings of the envp parameter are used.

**See Also:** Oracle Database SQL Language Reference for information on TO_DATE
**getDateTime()**

Sets Timestamp object to value represented by a string.

```cpp
void fromText(
    const string &timestampStr,
    const string &fmt,
    const string &nlsParam = "",
    const Environment *env = NULL);
```

Sets Timestamp object to value represented by a UString; globalization enabled.

```cpp
void fromText(
    const UString &timestampStr,
    const UString &fmt,
    const UString &nlsParam,
    const Environment *env = NULL);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestampStr</td>
<td>The timestamp string or UString to be converted to a Timestamp object.</td>
</tr>
<tr>
<td>fmt</td>
<td>The format string.</td>
</tr>
<tr>
<td>nlsParam</td>
<td>The NLS parameters string. If nlsParam is specified, this determines the NLS parameters to be used for the conversion. If nlsParam is not specified, the NLS parameters are picked up from envp.</td>
</tr>
<tr>
<td>env</td>
<td>The OCCI environment. In globalization enabled version of the method, used to determine NLS_CALENDAR for interpreting timestampStr. If env is not passed, the environment associated with the object controls the setting. Should be a non-NULL value if called on a NULL Timestamp object.</td>
</tr>
</tbody>
</table>

**getDateTime()**

Returns the year, month and day values of the Timestamp.

**Syntax**

```cpp
void getDateTime(
    int &year,
    unsigned int &month,
    unsigned int &day) const;
```

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>Year component.</td>
</tr>
<tr>
<td>month</td>
<td>Month component.</td>
</tr>
<tr>
<td>day</td>
<td>Day component.</td>
</tr>
</tbody>
</table>

**getTime()**

Returns the hour, minute, second, and fractional second components.

**Syntax**

```cpp
void getTime(
    unsigned int &hour,
    unsigned int &minute,
    unsigned int &second,
    unsigned int &fs) const;
```
getTimeZoneOffset()

Returns the time zone offset in hours and minutes.

Syntax

```cpp
void getTimeZoneOffset(
    int &hour,
    int &minute) const;
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hour</td>
<td>Hour component.</td>
</tr>
<tr>
<td>minute</td>
<td>Minute component.</td>
</tr>
<tr>
<td>second</td>
<td>Second component.</td>
</tr>
<tr>
<td>fs</td>
<td>Fractional second component.</td>
</tr>
</tbody>
</table>

intervalAdd()

Adds an interval to timestamp.

Syntax

```cpp
const Timestamp intervalAdd(
    const IntervalDS& val) const;
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Interval to be added.</td>
</tr>
</tbody>
</table>

intervalSub()

Subtracts an interval from a timestamp and returns the result as a timestamp. Returns a `Timestamp` with the value of this - `val`.

Syntax

```cpp
const Timestamp intervalSub(
    const IntervalDS& val) const;
```

```cpp
const Timestamp intervalsUB(
    const IntervalYM& val) const;
```

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Interval to be added.</td>
</tr>
</tbody>
</table>
isNull()

Returns TRUE if Timestamp is NULL or FALSE otherwise.

**Syntax**

```cpp
bool isNull() const;
```

**operator=()**

Assigns a given timestamp object to this object.

**Syntax**

```cpp
Timestamp & operator=(
    const Timestamp &src);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>Value to be assigned.</td>
</tr>
</tbody>
</table>

**operator==()**

Compares the timestamps specified. If the timestamps are equal, returns TRUE, FALSE otherwise. If either a or b is NULL then FALSE is returned.

**Syntax**

```cpp
bool operator==(const Timestamp &first, const Timestamp &second);
```

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>First timestamp to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>Second timestamp to be compared.</td>
</tr>
</tbody>
</table>

**operator!=()**

Compares the timestamps specified. If the timestamps are not equal then TRUE is returned; otherwise, FALSE is returned. If either timestamp is NULL then FALSE is returned.

**Syntax**

```cpp
bool operator!=(const Timestamp &first, const Timestamp &second);
```
Returns TRUE if first is greater than second, FALSE otherwise. If either is NULL then FALSE is returned.

**Syntax**

```cpp
bool operator>(
    const Timestamp &first,
    const Timestamp &second);
```

**operator>=()**

Compares the timestamps specified. If the first timestamp is greater than or equal to the second timestamp then TRUE is returned; otherwise, FALSE is returned. If either timestamp is NULL then FALSE is returned.

**Syntax**

```cpp
bool operator>=(
    const Timestamp &first,
    const Timestamp &second);
```

**operator<()**

Returns TRUE if first is less than second, FALSE otherwise. If either a or b is NULL then FALSE is returned.

**Syntax**

```cpp
bool operator<(
    const Timestamp &first,
    const Timestamp &second);
```
operator<=(())

Compares the timestamps specified. If the first timestamp is less than or equal to the second timestamp then TRUE is returned; otherwise, FALSE is returned. If either timestamp is NULL then FALSE is returned.

**Syntax**

```cpp
bool operator<=(
    const Timestamp &first,
    const Timestamp &second);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>First timestamp to be compared.</td>
</tr>
<tr>
<td>second</td>
<td>Second timestamp to be compared.</td>
</tr>
</tbody>
</table>

**setDate()**

Sets the year, month, day components contained for this timestamp.

**Syntax**

```cpp
void setDate(
    int year,
    unsigned int month,
    unsigned int day);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>Year component. Valid values are -4713 through 9999.</td>
</tr>
<tr>
<td>month</td>
<td>Month component. Valid values are 1 through 12.</td>
</tr>
<tr>
<td>day</td>
<td>Day component. Valid values are 1 through 31.</td>
</tr>
</tbody>
</table>

**setNull()**

Sets the timestamp to NULL.

**Syntax**

```cpp
void setNull();
```

**setTime()**

Sets the day, hour, minute, second and fractional second components for this timestamp.

**Syntax**

```cpp
void setTime(
```
unsigned int hour,
unsigned int minute,
unsigned int second,
unsigned int fs);  

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hour</td>
<td>Hour component. Valid values are 0 through 23.</td>
</tr>
<tr>
<td>minute</td>
<td>Minute component. Valid values are 0 through 59.</td>
</tr>
<tr>
<td>second</td>
<td>Second component. Valid values are 0 through 59.</td>
</tr>
<tr>
<td>fs</td>
<td>Fractional second component.</td>
</tr>
</tbody>
</table>

**setTimeZoneOffset()**

Sets the hour and minute offset for time zone.

**Syntax**

```c
void setTimeZoneOffset(
    int hour,
    int minute);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hour</td>
<td>Time zone hour. Valid values are -12 through 12.</td>
</tr>
<tr>
<td>minute</td>
<td>Time zone minute. Valid values are -59 through 59.</td>
</tr>
</tbody>
</table>

**subDS()**

Computes the difference between this timestamp and the specified timestamp and return the difference as an IntervalDS.

**Syntax**

```c
const IntervalDS subDS(
    const Timestamp& val) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Timestamp to be subtracted.</td>
</tr>
</tbody>
</table>

**subYM()**

Computes the difference between timestamp values and return the difference as an IntervalYM.

**Syntax**

```c
const IntervalYM subYM(
    const Timestamp& val) const;
```
toText()

Returns a string or UString representation for the timestamp in the format specified.

If nlsParam is specified, this determines the NLS parameters used for the conversion. If nlsParam is not specified, the NLS parameters are picked up from the environment associated with the instance, if any.

See Also: Oracle Database SQL Language Reference for information on TO_DATE

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Timestamp to be subtracted.</td>
</tr>
</tbody>
</table>

Syntax Description

string toText(
    const string &fmt,
    unsigned int fsprec,
    const string &nlsParam = "") const;

Returns a string representation for the timestamp in the format specified.

UString toText(
    const UString &fmt,
    unsigned int fsprec,
    const UString &nlsParam) const;

Returns a UString representation for the timestamp in the format specified; globalization enabled.

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt</td>
<td>The format string.</td>
</tr>
<tr>
<td>fsprec</td>
<td>The precision for the fractional second component of Timestamp.</td>
</tr>
<tr>
<td>nlsParam</td>
<td>The NLS parameters string. If nlsParam is specified, this determines the NLS parameters to be used for the conversion. If nlsParam is not specified, the NLS parameters are picked up from envp.</td>
</tr>
</tbody>
</table>
A
ADR, 12-12
  ADRC utility, 12-14
  base location, 12-13
ADR Command Interpreter, 12-12
ADRCI, 12-12, 12-14
Agent class, 13-7
  methods, 13-7
AnyData class, 13-10
  methods, 13-11
  supported data type, 13-11
  usage examples, 13-10
application managed data buffering, 12-8
application-provided serialization, 12-7
associative access
  overview, 4-10
atomic null, 4-18
attributes, 1-7
automatic diagnostic repository (ADR), 12-12
automatic serialization, 12-7

B
BatchSQLException
  methods, 13-19
BatchSQLException class, 13-19
Bfile class, 13-20
  methods, 13-20
BFILEs
  external data type, 5-6
bind operations
  in bind operations, 1-7
  out bind operations, 1-7
Blob class, 13-26
  methods, 13-26
BLOBs
  external data type, 5-7
Bytes class, 13-35
  methods, 13-35

C
callable statements, 3-14
  with arrays as parameters, 3-15
CASE OTT parameter, 8-5
CHAR
  external data type, 5-7
classes
  Agent class, 13-7
  AnyData class, 13-10
  BatchSQLException class, 13-19
  Bfile class, 13-20
  Blob class, 13-26
  Bytes class, 13-35
  Clob class, 13-38
  Connection class, 13-49
  ConnectionPool class, 13-62
  Consumer class, 13-67
  Date class, 13-75
  Environment class, 13-85
  IntervalDS class, 13-97
  IntervalYM class, 13-107
  Listener class, 13-116
  Map class, 13-118
  Message class, 13-120
  Metadata class, 13-128
  NotifyResult class, 13-142
  Number class, 13-143
  PObject class, 13-162
  Producer class, 13-168
  Ref class, 13-173
  RefAny class, 13-178
  ResultSet class, 3-19, 13-181
  SQLException class, 13-200
  StatelessConnectionPool class, 13-203
  Statement class, 13-213
  Stream class, 13-254
  Subscription class, 13-257
  Timestamp class, 13-264
  Client Result Cache, 12-16
    hint, 12-16
  Clob class, 13-38
    methods, 13-39
  CLOBs
    external data type, 5-8
  CODE OTT parameter, 8-6
collections
  working with, 4-16
  committing a transaction, 3-23
complex object retrieval
  complex object, 4-14
UNSIGNED INT, 5-14
VARCHAR, 5-14
VARCHAR2, 5-14
VARNUM, 5-15
VARRAW, 5-15

F
Fault Diagnosability
disabling, 12-16
fault diagnosability, 12-12
fields
IntervalDS class, 13-97
IntervalYM class, 13-107
Timestamp fields, 13-264
FLOAT
external data type, 5-9

H
HFILE OTT parameter, 8-7

I
index-organized table, 5-3
Instant Client, 2-2
benefits, 2-2
collection names, 2-5
database connection, 2-5
environment variables, 2-5
Solaris, 2-6
Windows, 2-6
installation, 2-2
libraries, 2-4
Data Shared Library, 2-4
patching, 2-4
regenerating, 2-4
patching libraries, 2-4
SDK, 2-3
using, 2-4
Instant Client Light (English), 2-6
character sets, 2-6
ers, 2-6
globalization settings, 2-6
installation, 2-7
Client Admin Install, 2-7
Oracle Universal Installer, 2-7
OTN download, 2-7
using, 2-7
INTEGER
external data type, 5-9
internal data type, 5-2
CHAR, 5-3
LONG, 5-3
LONG RAW, 5-3
RAW, 5-3
VARCHAR2, 5-3
INTERVAL DAY TO SECOND
external data type, 5-9
INTERVAL YEAR TO MONTH
external data type, 5-10
IntervalDS class, 13-97
fields, 13-97
methods, 13-98
usage examples, 13-97
IntervalYM class, 13-107
fields, 13-107
methods, 13-108
usage examples, 13-107
INTYPE file
structure of, 8-13
INTYPE OTT parameter, 8-7

L
Listener class, 13-116
methods, 13-116
LOBs
external data type
data conversion, 5-17
LONG
external data type, 5-10
LONG RAW
external data type, 5-10
LONG VARCHAR
external data type, 5-10

M
manipulating object attributes, 4-12
Map class, 13-118
methods, 13-118
Message class, 13-120
methods, 13-120
metadata
argument and result attributes, 6-12
attribute groupings, 6-2
collection attributes, 6-2
column attributes, 6-2
database attributes, 6-3
list attributes, 6-2
package attributes, 6-2
parameter attributes, 6-2
procedure, function, and subprogram attributes, 6-2
schema attributes, 6-3
sequence attributes, 6-2
synonym attributes, 6-2
table and view attributes, 6-2
type attribute attributes, 6-2
type attributes, 6-2
type method attributes, 6-2
attributes, 6-6
code example, 6-3
collection attributes, 6-10
column attributes, 6-12
database attributes, 6-14
listing database objects, 6-2
list attributes, 6-13
overview, 6-1
package attributes, 6-7
parameter attributes, 6-6
procedure, function, and subprogram attributes, 6-7
schema attributes, 6-14
sequence attributes, 6-11
synonym attributes, 6-11
table and view attributes, 6-6
type attribute attributes, 6-9
type attributes, 6-8
type methods attributes, 6-9
MetaData class, 13-128
methods, 13-136
methods, 1-7
Agent methods, 13-7
AnyData methods, 13-11
BatchSQLException methods, 13-19
Bfile methods, 13-20
Blob methods, 13-26
Bytes methods, 13-35
Clob methods, 13-38
Connection methods, 13-49
ConnectionPool methods, 13-62
Consumer methods, 13-67
createConnection method, 3-2
createConnectionPool method, 3-3
createEnvironment method, 3-2
createProxyConnection method, 3-4
createStatement method, 3-12
Date methods, 13-75
Environment class, 13-85
execute method, 3-12
executeArrayUpdate method, 3-12, 12-10
executeQuery method, 3-12
IntervalDS methods, 13-98
IntervalYM class, 13-108
Listener methods, 13-116
Map methods, 13-118
Message methods, 13-120
MetaData class, 13-136
NotifyResult methods, 13-142
Number class, 13-144
PObject methods, 13-162
Producer methods, 13-168
Ref methods, 13-173
RefAny methods, 13-178
ResultSet methods, 13-181
setDataBuffer method, 12-9
SQLException methods, 13-200
StatelessConnectionPool, 13-203
Statement, 13-213
Stream methods, 13-254
Subscription methods, 13-257
terminateConnection method, 3-2, 3-3
terminateEnvironment method, 3-3
terminateStatement method, 3-13
Timestamp methods, 13-264
modifying rows iteratively, 12-11

N
navigational access
overview, 4-10
NCLOBs
external data type, 5-11
NEEDS_STREAM_DATA status, 3-22, 3-23
nonprocedural elements, 1-3
nonreferenceable objects, 4-2
NotifyResult class, 13-142
methods, 13-142
nullness, 4-17
NUMBER
external data type, 5-11
Number class, 13-143
methods, 13-144
usage examples, 13-143

O
object cache, 4-8
flushing, 4-9
object mode, 4-7
object programming
overview, 4-1
using OCCI, 4-1
object references
using, 4-18
see also REF
Object Type Translator utility
see OTT utility
object types, 1-7
objects
access using SQL, 4-10
attributes, 1-7
client-side, 1-8
dirty, 4-12
environment, 1-8
flushing, 4-12
freeing, 4-18
in OCCI, 4-2
inserting, 4-10
interfaces, 1-8
associative, 1-8
navigational, 1-8
manipulating attributes, 4-12
marking, 4-12
Metadata Class, 1-9
methods, 1-7
modifying, 4-10
object cache, 1-8
object types, 1-7
pinned, 4-11
pinning, 4-8, 4-11
recording database changes, 4-12
runtime environment, 1-8
transparent application failover, 12-3
OCCI
benefits, 1-2
building applications, 1-2
functionality, 1-3
object mode, 4-7
overview, 1-1
special SQL terms, 1-6

OCCI classes
Agent class, 13-7
AnyData class, 13-10
BatchSQLException class, 13-19
Bfile class, 13-20
Blob class, 13-26
Bytes class, 13-35
Clob class, 13-38
Connection class, 13-49
ConnectionPool class, 13-62
Consumer class, 13-67
Data class, 13-75
Environment class, 13-85
IntervalDS class, 13-97
IntervalYM class, 13-107
Listener class, 13-116
Map class, 13-118
Message class, 13-120
MetaData class, 13-128
NotifyResult class, 13-142
Number class, 13-143
PObject class, 13-162
Producer class, 13-168
Ref class, 13-173
RefAny class, 13-178
ResultSet class, 13-181
SQLException class, 13-200
StatelessConnectionPool class, 13-203
Statement class, 13-213
Stream class, 13-254
Subscription class, 13-257
Timestamp class, 13-264

OCCI environment
connection pool, 3-3
creating, 3-1
opening a connection, 3-2
scope, 3-2
terminating, 3-1

OCCI program
element of, 4-21
OCCI program development, 4-6
operational flow, 4-7
program structure, 4-6

OCCI types
data conversion, 5-1
optimizing performance, 3-20
setting prefetch count, 3-20

OTT parameter TRANSITIVE, 8-9
OTT parameters
CASE, 8-5
CONFIG, 8-6
OUTTYPE, 8-7
SCHEMA_NAMES, 8-8

USERID, 8-11
where they appear, 8-11

OTT utility
benefits, 1-9
creating types in the database, 8-2
default name mapping, 8-22
description, 1-9
parameters, 8-4
using, 8-2
out bind variables, 1-6

OUTTYPE OTT parameter, 8-7

parameterized statements, 3-13

performance
optimizing
executeArrayUpdate method, 12-10
setDateBuffer method, 12-9

performance tuning, 12-1
application managed data buffering, 12-8
array fetch using next() method, 12-11
connection sharing, 12-5
data buffering, 12-8
reading and writing multiple LOBs, 7-7
shared server environments, 12-6
thread safety, 12-6

thread safety, 12-6
transparent application failover, 12-1

persistent objects, 4-2
creating, 4-4
standalone objects, 4-3
types
embedded objects, 4-2
nonreferenceable objects, 4-2
referenceable objects, 4-2
standalone objects, 4-2

pinning objects, 4-8, 4-11

PL/SQL
out bind variables, 1-6
overview, 1-5

PObject class, 13-162
methods, 13-162

prefetch count
set, 3-20

prefetch limit, 4-14

PREPARED status, 3-21, 3-22

procedural elements, 1-3

Producer class, 13-168
methods, 13-168

proxy connections, 3-4
using createProxyConnection method, 3-4

queries, 1-5
how to specify, 3-20

RAW
external data type, 5-13
REF external data type, 5-13
retrieving a reference to an object
Ref class, 13-173
methods, 13-173
RefAny class, 13-178
methods, 13-178
referenceable objects, 4-2
relational programming
using OCCI, 3-1
RESULT_SET_AVAILABLE status, 3-21, 3-22
ResultSet class, 3-19, 13-181
methods, 13-181
root object, 4-14
ROWID external data type, 5-13
rows
iterative modification, 12-11
modify, 12-11
S
SCHEMA_NAMES OTT parameter, 8-8
shared connections
using, 12-5
shared server environments
application-provided serialization, 12-7
automatic serialization, 12-7
concurrency, 12-7
thread safety, 12-6
implementing, 12-6
SQL statements
control statements, 1-5
DML statements, 1-5
processing of, 1-4
queries, 1-5
types
callable statements, 3-13, 3-14
parameterized statements, 3-13
standard statements, 3-13
SQLException class, 13-200
methods, 13-200
sqlnet.ora, 12-16
standalone objects, 4-2
creating, 4-3
standard statements, 3-13
StatelessConnectionPool class, 13-203
methods, 13-203
status
NEEDS_STREAM_DATA, 3-22, 3-23
PREPARED, 3-21, 3-22
RESULT_SET_AVAILABLE, 3-21, 3-22
STREAM_DATA_AVAILABLE, 3-22, 3-23
UNPREPARED, 3-21, 3-22
UPDATE_COUNT_AVAILABLE, 3-22
Stream class, 13-254
methods, 13-254
STREAM_DATA_AVAILABLE status, 3-22, 3-23
streamed reads, 3-15
streamed writes, 3-15
STRING
external data type, 5-13
Subscription class, 13-257
methods, 13-257
substitutability, 4-19
T
table
index-organized, 5-3
thread safety, 12-6
implementing, 12-6
TIMESTAMP
external data type, 5-13
Timestamp class
methods, 13-264
TIMESTAMP WITH LOCAL TIME ZONE external data type, 5-14
TIMESTAMP WITH TIME ZONE external data type, 5-14
transient objects, 4-2, 4-3
creating, 4-4, 4-5
TRANSITIVE OTT parameter, 8-9
transparent application failover, 12-1
connection pooling, 12-3
objects, 12-3
using, 12-2
type inheritance, 4-18, 4-20
U
UNPREPARED status, 3-21, 3-22
UNSIGNED INT external data type, 5-14
UPDATE_COUNT_AVAILABLE status, 3-22
USERID OTT parameter, 8-11
V
values
in context of this document, 4-4
in object applications, 4-4
VARCHAR external data type, 5-14
VARCHAR2 external data type, 5-14
VARNUM external data type, 5-15
VARRAW external data type, 5-11, 5-15
XA library, 11-1