Oracle® AI Database

Oracle® C++ Call Interface Developer's Guide

26ai
G43218-01
October 2025

Oracle AI Database Oracle® C++ Call Interface Developer's Guide, 26ai

G43218-01

Copyright © 1999, 2025, Oracle and/or its affiliates.

Primary Author: Mamata Basapur

Contributing Authors: Rod Ward

Contributors: Sandeepan Banerjee, Subhranshu Banerjee, Kalyanji Chintakayala, Krishna Itikarlapalli, Shankar Iyer, Maura Joglekar, Toliver Jue, Ravi Kasamsetty, Srinath Krishnaswamy, Shoaib Lari, Geoff Lee, Roza Leyderman, Chetan Maiya, Kuassi Mensah, Vipul Modi, Sue Pelski, Rajendra Pingte, John Stewart, Rekha Vallam, Krishna Verma, Alan Willaims

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience i
Related Documents i
Conventions ii

1 Introduction to OCCI

1.1 Changes in This Release for Oracle C++ Call Interface Programmer's Guide 1
1.2 Overview of OCCI 2
1.2.1 About Benefits of OCCI 2
1.2.2 About Building a C++ Application with OCCI 3
1.2.3 About Functionality of OCCI 4
1.2.4 About Procedural and Nonprocedural Elements 4
1.3 About Processing SQL Statements 5
1.3.1 About Data Definition Language Statements 5
1.3.2 About Control Statements 5
1.3.3 About Data Manipulation Language Statements 6
1.3.4 About Queries 6
1.4 Overview of PL/SQL 6
1.5 About Special OCCI/SQL Terms 7
1.6 About Object Support 8
1.6.1 About Client-Side Object Cache 8
1.6.2 About Run-time Environment for Objects 9
1.6.3 About Associative and Navigational Interfaces 9
1.6.4 About Interoperability with C (OCI) 10
1.6.5 About the Metadata Class 10
1.6.6 About the Object Type Translator Utility 10
1.7 About Additional Support 11
1.7.1 Building OCCI Demos 11
1.7.2 About OCCI on the Oracle Technology Network 11

2 Installation and Upgrading

2.1 About Installing Oracle C++ Call Interface 1

Copyright © 1999, 2025, Oracle and/or its affiliates.

2.2 About Upgrading Considerations 1
2.3 About Determining the Oracle Database Versions 1
2.3.1 Determining the Oracle Client Version During Compilation 1
2.3.2 About Determining the Oracle Client and Server Versions at Run Time 2
2.4 Oracle Instant Client and Oracle Instant client Basic Light 2
2.5 About Using OCCI with Microsoft Visual C++ 2

3 Accessing Oracle AI Database Using C++

3.1 About Connecting to a Database 1
3.1.1 Creating and Terminating an Environment 1
3.1.2 Opening and Closing a Connection 2
3.1.3 About Support for Pluggable Databases 3
3.2 About Pooling Connections 3
3.2.1 About Using Connection Pools 3
3.2.1.1 Creating a Connection Pool 4
3.2.1.2 Creating Proxy Connections 5
3.2.2 Using Stateless Connection Pooling 5
3.2.3 About Database Resident Connection Pooling 9
3.2.3.1 Administrating Database Resident Connection Pools 10
3.2.3.2 Using Database Resident Connection Pools 11
3.3 About Executing SQL DDL and DML Statements 12
3.3.1 Creating a Statement Object 12
3.3.2 Creating a Statement Object that Executes SQL Commands 12
3.3.2.1 Creating a Database Table 13
3.3.2.2 Inserting Values into a Database Table 13
3.3.3 Reusing the Statement Object 13
3.3.4 Terminating a Statement Object 14
3.4 About Types of SQL Statements in the OCCI Environment 14
3.4.1 About Standard Statements 14
3.4.2 Using Parameterized Statements 14
3.4.3 Using Callable Statements 15
3.4.3.1 Using Callable Statements that Use Array Parameters 15
3.4.4 About Streamed Reads and Writes 16
3.4.4.1 Binding Data in Streaming Mode; SELECT/DML and PL/SQL 17
3.4.4.2 Fetching Data in Streaming Mode: PL/SQL 17
3.4.4.3 About Fetching Data in Streaming Mode: ResultSet 18
3.4.4.4 Working with Multiple Streams 18
3.4.5 About Modifying Rows Iteratively 19
3.4.5.1 Setting the Maximum Number of Iterations 19
3.4.5.2 Setting the Maximum Parameter Size 19
3.4.5.3 Executing an Iterative Operation 20

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.5 About Executing SQL Queries 20
3.5.1 Using the Result Set 20
3.5.2 About Specifying the Query 21
3.5.3 About Optimizing Performance by Setting Prefetch Count 22
3.6 About Executing Statements Dynamically 22
3.6.1 About Statement Status Definitions 23
3.6.1.1 UNPREPARED 23
3.6.1.2 PREPARED 23
3.6.1.3 RESULT_SET_AVAILABLE 23
3.6.1.4 UPDATE_COUNT_AVAILABLE 24
3.6.1.5 NEEDS_STREAM_DATA 24
3.6.1.6 STREAM_DATA_AVAILABLE 24
3.7 About Using Larger Row Count and Error Code Range Data Types 25
3.7.1 Using Larger Row Count in SELECT Operations 25
3.7.2 Using Larger Row Count in INSERT, UPDATE, and DELETE Operations 26
3.8 About Committing a Transaction 27
3.9 Caching Statements 27
3.10 About Handling Exceptions 29
3.10.1 About Handling Null and Truncated Data 30

4 Object Programming

4.1 Overview of Object Programming 1
4.2 About Working with Objects in C++ with OCCI 2
4.2.1 About Persistent Objects 2
4.2.2 About Transient Objects 3
4.2.3 About Values 4
4.3 About Representing Objects in C++ Applications 4
4.3.1 Creating Persistent and Transient Objects 4
4.3.2 Creating Object Representations using the OTT Utility 5
4.4 About Developing a C++ Application using OCCI 5
4.4.1 Developing Basic Object Program Structure 6
4.4.2 About Basic Object Operational Flow 7
4.4.2.1 About Initializing OCCI in Object Mode 7
4.4.2.2 About Pinning anObject 8
4.4.2.3 About Operating on an Object in Cache 8
4.4.2.4 About Flushing Changes to the Object 8
4.4.2.5 About Deletion of an Object 9
4.5 Migrating C++ Applications to Oracle Using OCCI 9
4.6 Overview of Associative Access 9
4.6.1 Using SQL to Access Objects 10
4.6.2 Inserting and Modifying Values 10

Copyright © 1999, 2025, Oracle and/or its affiliates.

4.7 Overview of Navigational Access 10
4.7.1 Retrieving an Object Reference (REF) from the Database Server 11
4.7.2 Pinning an Object 11
4.7.3 Manipulating Object Attributes 12
4.7.4 About Marking Objects and Flushing Changes 12
4.7.5 Marking an Object as Modified (Dirty) 12
4.7.6 About Recording Changes in the Database 13
4.7.7 Collecting Garbage in the Object Cache 13
4.7.8 About Ensuring Transactional Consistency of References 14
4.8 Overview of Complex Object Retrieval 14
4.8.1 Retrieving Complex Objects 15
4.8.2 About Prefetching Complex Objects 16
4.9 Working with Collections 16
4.9.1 Fetching Embedded Objects 17
4.9.2 About Nullness 18
4.10 About Using Object References 18
4.11 About Deleting Objects from the Database 18
4.12 About Type Inheritance 19
4.12.1 About Substitutability 20
4.12.2 Declaring NOT INSTANTIABLE Types and Methods 20
4.12.3 About OCCI Support for Type Inheritance 21
4.12.3.1 About Connection::getMetaData() 21
4.12.3.2 About Bind and Define Functions 21
4.12.4 About OTT Support for Type Inheritance 21
4.13 A Sample OCCI Application 22

5 Data Types

5.1 Overview of Oracle Data Types 1
5.1.1 About OCCI Type and Data Conversion 1
5.2 Internal Data Types 2
5.2.1 Character Strings and Byte Arrays 3
5.2.2 Universal Rowid (UROWID) 4
5.3 External Data Types 4
5.3.1 Description of External Data Types 8
5.3.1.1 BFILE 9
5.3.1.2 BDOUBLE 9
5.3.1.3 BFLOAT 9
5.3.1.4 BLOB 10
5.3.1.5 CHAR 10
5.3.1.6 CHARZ 10
5.3.1.7 CLOB 10

Copyright © 1999, 2025, Oracle and/or its affiliates.

5.3.1.8 DATE 10
5.3.1.9 FLOAT 11
5.3.1.10 INTEGER 12
5.3.1.11 INTERVAL DAY TO SECOND 12
5.3.1.12 INTERVAL YEAR TO MONTH 12
5.3.1.13 LONG 13
5.3.1.14 LONG RAW 13
5.3.1.15 LONG VARCHAR 13
5.3.1.16 LONG VARRAW 13
5.3.1.17 NCLOB 13
5.3.1.18 NUMBER 13
5.3.1.19 OCCI BFILE 14
5.3.1.20 OCCI BLOB 14
5.3.1.21 OCCIBOOL 14
5.3.1.22 OCCI BYTES 14
5.3.1.23 OCCI CLOB 15
5.3.1.24 OCCI DATE 15
5.3.1.25 OCCI INTERVALDS 15
5.3.1.26 OCCI INTERVALYM 15
5.3.1.27 OCCI NUMBER 15
5.3.1.28 OCCI POBJECT 15
5.3.1.29 OCCI REF 16
5.3.1.30 OCCI REFANY 16
5.3.1.31 OCCI STRING 16
5.3.1.32 OCCI TIMESTAMP 16
5.3.1.33 OCCI VECTOR 16
5.3.1.34 RAW 16
5.3.1.35 REF 17
5.3.1.36 ROWID 17
5.3.1.37 STRING 17
5.3.1.38 TIMESTAMP 17
5.3.1.39 TIMESTAMP WITH LOCAL TIME ZONE 17
5.3.1.40 TIMESTAMP WITH TIME ZONE 18
5.3.1.41 UNSIGNED INT 18
5.3.1.42 VARCHAR 18
5.3.1.43 VARCHAR2 18
5.3.1.44 VARNUM 18
5.3.1.45 VARRAW 19
5.3.1.46 NATIVE DOUBLE 19
5.3.1.47 NATIVE FLOAT 19
5.4 Data Conversions 19
5.4.1 Data Conversions for LOB Data Types 21

Copyright © 1999, 2025, Oracle and/or its affiliates.

5.4.2 Data Conversions for Date, Timestamp, and Interval Data Types 22

6 Metadata

6.1 Overview of Metadata 1
6.2 Using Identity Column Metadata 2
6.3 About Describing Database Metadata 3
6.3.1 Using Metadata (Code Examples) 4
6.4 Attribute Reference Information 8
6.4.1 Parameter Attributes 8
6.4.2 Table and View Attributes 9
6.4.3 Procedure, Function, and Subprogram Attributes 10
6.4.4 Package Attributes 10
6.4.5 Type Attributes 11
6.4.6 Type Attribute Attributes 12
6.4.7 Type Method Attributes 13
6.4.8 Collection Attributes 14
6.4.9 Synonym Attributes 15
6.4.10 Sequence Attributes 15
6.4.11 Column Attributes 15
6.4.12 Argument and Result Attributes 16
6.4.13 List Attributes 18
6.4.14 Schema Attributes 18
6.4.15 Database Attributes 18

7 Programming with LOBs

7.1 Overview of LOBs 1
7.1.1 Introducing Internal LOBs 1
7.1.2 Introducing External LOBs 2
7.1.3 About Storing LOBs 2
7.2 Creating LOBs in OCCI Applications 2
7.3 Restricting the Opening and Closing of LOBs 3
7.4 About Reading and Writing LOBs 3
7.4.1 Reading LOBs 4
7.4.2 Writing LOBs 5
7.4.3 About Enhancing the Performance of LOB Reads and Writes 6
7.4.3.1 About Using the getChunkSize() Method 7
7.4.4 Updating LOBs 7
7.4.5 About Reading and Writing Multiple LOBs 7
7.4.5.1 About Using the Interfaces for Reading and Writing Multiple LOBs 8
7.5 Using Objects with LOB Attributes 8

Copyright © 1999, 2025, Oracle and/or its affiliates.

7.6 About Using SecureFiles 9
7.6.1 About Using SecureFile Compression 10
7.6.2 About Using SecureFiles Encryption 10
7.6.3 About Using SecureFiles Deduplication 10
7.6.4 About Combining SecureFiles Compression, Encryption, and Deduplication 10
7.6.5 SecureFiles LOB Types and Constants 10

8 Object Type Translator Utility

8.1 Overview of the Object Type Translator Utility 1
8.2 Using the OTT Utility 2
8.3 Creating Types in the Database 2
8.4 About Invoking the OTT Utility 2
8.4.1 Specifying OTT Parameters 3
8.4.1.1 About Setting Parameters on the Command Line 3
8.4.1.2 About Setting Parameters in the INTYPE File 3
8.4.1.3 About Setting Parameters in the Configuration File 4
8.4.2 Invoking the OTT Utility on the Command Line 4
8.4.2.1 Elements Used on the OTT Command Line 5
8.4.3 OTT Utility Parameters 5
8.4.3.1 ATTRACCESS 6
8.4.3.2 CASE 6
8.4.3.3 CODE 7
8.4.3.4 CONFIG 7
8.4.3.5 CPPFILE 7
8.4.3.6 ERRTYPE 7
8.4.3.7 HFILE 7
8.4.3.8 INTYPE 8
8.4.3.9 MAPFILE 8
8.4.3.10 MAPFUNC 8
8.4.3.11 OUTTYPE 8
8.4.3.12 SCHEMA_NAMES 9
8.4.3.13 TRANSITIVE 10
8.4.3.14 UNICODE 10
8.4.3.15 USE_MARKER 12
8.4.3.16 USERID 12
8.4.4 Where OTT Parameters Can Appear 12
8.4.5 File Name Comparison Restriction 13
8.4.6 Using the OTT Command on Microsoft Windows 13
8.5 About Using the INTYPE File 13
8.5.1 Using the INTYPE File 14
8.5.2 Structure of the INTYPE File 15

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.5.2.1 INTYPE File Type Specifications 15
8.5.3 Using Nested include File Generation 16
8.6 Using OTT Utility Data Type Mappings 19
8.6.1 Default Name Mapping 23
8.7 Overview of the OUTTYPE File 24
8.8 Using the OTT Utility and OCCI Applications 25
8.9 Generating C++ Classes Generated by the OTT Utility 27
8.9.1 Map Registry Function 28
8.9.2 Extending C++ Classes 28
8.9.3 Carrying Forward User Added Code 29
8.9.3.1 How to Use Properties of OTT Markers 29
8.9.3.2 Using OTT Markers 30

9 Globalization and Unicode Support

9.1 Overview of Globalization and Unicode Support 1
9.2 Specifying Charactersets 1
9.3 Data Types for Globalization and Unicode Support 2
9.3.1 Using the UString Data Type 2
9.3.2 Using Multibyte and UTF16 data 2
9.3.3 Using CLOB and NCLOB Data Types 3
9.4 About Using Objects and OTT Support 3

10 Oracle Database Advanced Queuing

10.1 Overview of Oracle Database Advanced Queuing 1
10.2 About AQ Implementation in OCCI 2
10.2.1 Message 2
10.2.2 Agent 3
10.2.3 Producer 3
10.2.4 Consumer 3
10.2.5 Listener 4
10.2.6 Subscription 4
10.3 About Creating Messages 4
10.3.1 About Message Payloads 4
10.3.1.1 RAW 4
10.3.1.2 AnyData 4
10.3.1.3 Using User-defined Types as Payloads 5
10.3.2 Message Properties 5
10.3.2.1 Correlation 5
10.3.2.2 Sender 5
10.3.2.3 Delay and Expiration 6

Copyright © 1999, 2025, Oracle and/or its affiliates.

10.3.2.4 Recipients 6
10.3.2.5 Priority and Ordering 6
10.4 Enqueuing Messages 6
10.5 Dequeuing Messages 7
10.5.1 About Dequeuing Options 7
10.5.1.1 Correlation 7
10.5.1.2 Mode 8
10.5.1.3 Navigation 8
10.6 Listening for Messages 8
10.7 About Registering for Notification 8
10.7.1 Publish-Subscribe Notifications 9
10.7.1.1 How to Use Direct Registration 9
10.7.1.2 Using Open Registration 10
10.7.2 About Notification Callback 11
10.8 About Message Format Transformation 12

11 Oracle XA Library

11.1 Developing Applications with XA and OCCI 1
11.2 APIs for XA Support 2

12 Optimizing Performance of C++ Applications

12.1 About Transparent Application Failover 1
12.1.1 Using Transparent Application Failover 2
12.1.2 About Objects and Transparent Application Failover 3
12.1.3 Using Connection Pooling and Transparent Application Failover 3
12.2 About Connection Sharing 5
12.2.1 Introduction to Thread Safety 6
12.2.2 Implementing Thread Safety 6
12.2.3 About Serialization 7
12.2.3.1 Automatic Serialization 7
12.2.3.2 Application-Provided Serialization 7
12.2.4 Operating System Considerations 8
12.3 About Application-Managed Data Buffering 8
12.3.1 Using the setDataBuffer() Method 9
12.3.2 Using the executeArrayUpdate() Method 10
12.4 Using the Array Fetch Using next() Method 11
12.5 Modifying Rows Iteratively 11
12.6 About Using Oracle Connection Manager in Traffic Director Mode 12
12.7 About Run-time Load Balancing of the Stateless Connection Pool 14
12.7.1 API Support 15

Copyright © 1999, 2025, Oracle and/or its affiliates.

12.8 About Fault Diagnosability 15
12.8.1 Using ADR Base Location 15
12.8.2 Using ADRCI 17
12.8.3 Controlling ADR Creation and Disabling Fault Diagnosability 19
12.9 Using Client Result Cache 19
12.10 About Client-Side Deployment Parameters and Auto Tuning 20
12.10.1 Support for Centralized Configuration Store 20
12.10.1.1 Examples for Centralized Configuration Store OCCI Support 20

13 OCCI Application Programming Interface

13.1 OCCI Classes and Methods 1
13.1.1 Using OCCI Classes 2
13.1.2 OCCI Support for Windows NT and z/OS 3
13.1.2.1 Working with Collections of Refs 3
13.2 Common OCCI Constants 5
13.3 Agent Class 5
13.3.1 Agent() 5
13.3.2 getAddress() 6
13.3.3 getName() 6
13.3.4 getProtocol() 6
13.3.5 isNull() 7
13.3.6 operator=() 7
13.3.7 setAddress() 7
13.3.8 setName() 7
13.3.9 setNull() 8
13.3.10 setProtocol() 8
13.4 AnyData Class 8
13.4.1 AnyData() 11
13.4.2 getAsBDouble() 11
13.4.3 getAsBfile() 11
13.4.4 getAsBFloat() 12
13.4.5 getAsBytes() 12
13.4.6 getAsDate() 12
13.4.7 getAsIntervalDS() 12
13.4.8 getAsIntervalYM() 12
13.4.9 getAsNumber() 12
13.4.10 getAsObject() 12
13.4.11 getAsRef() 13
13.4.12 getAsString() 13
13.4.13 getAsTimestamp() 13
13.4.14 getType() 13

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.4.15 isNull() 13
13.4.16 setFromBDouble() 13
13.4.17 setFromBfile() 14
13.4.18 setFromBFloat() 14
13.4.19 setFromBytes() 14
13.4.20 setFromDate() 14
13.4.21 setFromIntervalDS() 15
13.4.22 setFromIntervalYM() 15
13.4.23 setFromNumber() 15
13.4.24 setFromObject() 15
13.4.25 setFromRef() 16
13.4.26 setFromString() 16
13.4.27 setFromTimestamp() 16
13.4.28 setNull() 17
13.5 BatchSQLException Class 17
13.5.1 getException() 17
13.5.2 getFailedRowCount() 17
13.5.3 getRowNum() 18
13.6 Bfile Class 18
13.6.1 Bfile() 19
13.6.2 close() 20
13.6.3 closeStream() 20
13.6.4 fileExists() 20
13.6.5 getDirAlias() 20
13.6.6 getFileName() 20
13.6.7 getStream() 20
13.6.8 getUStringDirAlias() 21
13.6.9 getUStringFileName() 21
13.6.10 isInitialized() 21
13.6.11 isNull() 21
13.6.12 isOpen() 22
13.6.13 length() 22
13.6.14 open() 22
13.6.15 operator=() 22
13.6.16 operator==() 22
13.6.17 operator!=() 23
13.6.18 read() 23
13.6.19 setName() 23
13.6.20 setNull() 24
13.7 Blob Class 24
13.7.1 Blob() 25
13.7.2 append() 26

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.7.3 close() 26
13.7.4 closeStream() 26
13.7.5 copy() 27
13.7.6 getChunkSize() 27
13.7.7 getContentType() 27
13.7.8 getOptions() 28
13.7.9 getStream() 28
13.7.10 isInitialized() 28
13.7.11 isNull() 28
13.7.12 isOpen() 29
13.7.13 length() 29
13.7.14 open() 29
13.7.15 operator=() 29
13.7.16 operator==() 29
13.7.17 operator!= () 30
13.7.18 read() 30
13.7.19 setContentType() 30
13.7.20 setEmpty() 31
13.7.21 setNull() 31
13.7.22 setOptions() 31
13.7.23 trim() 32
13.7.24 write() 32
13.7.25 writeChunk() 32
13.8 Bytes Class 33
13.8.1 Bytes() 33
13.8.2 byteAt() 34
13.8.3 getBytes() 34
13.8.4 isNull() 35
13.8.5 length() 35
13.8.6 operator=() 35
13.8.7 setNull() 35
13.9 Clob Class 35
13.9.1 Clob() 37
13.9.2 append() 37
13.9.3 close() 38
13.9.4 closeStream() 38
13.9.5 copy() 38
13.9.6 getCharSetForm() 39
13.9.7 getCharSetId() 39
13.9.8 getCharSetIdUString() 39
13.9.9 getChunkSize() 39
13.9.10 getContentType() 40

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.9.11 getOptions() 40
13.9.12 getStream() 40
13.9.13 isInitialized() 40
13.9.14 isNull() 41
13.9.15 isOpen() 41
13.9.16 length() 41
13.9.17 open() 41
13.9.18 operator=() 41
13.9.19 operator==() 42
13.9.20 operator!=() 42
13.9.21 read() 42
13.9.22 setCharSetId() 43
13.9.23 setCharSetIdUString() 43
13.9.24 setCharSetForm() 44
13.9.25 setContentType() 44
13.9.26 setEmpty() 44
13.9.27 setNull() 45
13.9.28 setOptions() 45
13.9.29 trim() 45
13.9.30 write() 45
13.9.31 writeChunk() 46
13.10 Connection Class 47
13.10.1 changePassword() 48
13.10.2 commit() 49
13.10.3 createStatement() 49
13.10.4 flushCache() 50
13.10.5 getClientCharSet() 50
13.10.6 getClientCharSetUString() 50
13.10.7 getClientNCHARCharSet() 50
13.10.8 getClientNCHARCharSetUString() 50
13.10.9 getClientVersion() 51
13.10.10 getLTXID() 51
13.10.11 getMetaData() 51
13.10.12 getOCIServer() 52
13.10.13 getOCIServiceContext() 52
13.10.14 getOCISession() 52
13.10.15 getServerVersion() 52
13.10.16 getServerVersionUString() 53
13.10.17 getStmtCacheSize() 53
13.10.18 getTag() 53
13.10.19 isCached() 53
13.10.20 pinVectorOfRefs() 53

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.10.21 postToSubscriptions() 54
13.10.22 readVectorOfBfiles() 54
13.10.23 readVectorOfBlobs() 55
13.10.24 readVectorOfClobs() 56
13.10.25 registerSubscriptions() 56
13.10.26 rollback() 57
13.10.27 setStmtCacheSize() 57
13.10.28 setTAFNotify() 57
13.10.29 terminateStatement() 58
13.10.30 unregisterSubscription() 58
13.10.31 writeVectorOfBlobs() 59
13.10.32 writeVectorOfClobs() 59
13.11 ConnectionPool Class 60
13.11.1 createConnection() 61
13.11.2 createProxyConnection() 61
13.11.3 getBusyConnections() 62
13.11.4 getIncrConnections() 62
13.11.5 getMaxConnections() 63
13.11.6 getMinConnections() 63
13.11.7 getOpenConnections() 63
13.11.8 getPoolName() 63
13.11.9 getStmtCacheSize() 63
13.11.10 getTimeOut() 63
13.11.11 setErrorOnBusy() 64
13.11.12 setPoolSize() 64
13.11.13 setStmtCacheSize() 64
13.11.14 setTimeOut() 64
13.11.15 terminateConnection() 65
13.12 Consumer Class 65
13.12.1 Consumer() 67
13.12.2 getConsumerName() 68
13.12.3 getCorrelationId() 68
13.12.4 getDequeueMode() 68
13.12.5 getMessageIdToDequeue() 68
13.12.6 getPositionOfMessage() 68
13.12.7 getQueueName() 68
13.12.8 getTransformation() 68
13.12.9 getVisibility() 69
13.12.10 getWaitTime() 69
13.12.11 isNull() 69
13.12.12 operator=() 69
13.12.13 receive() 69

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.12.14 setAgent() 70
13.12.15 setConsumerName() 70
13.12.16 setCorrelationId() 70
13.12.17 setDequeueMode() 71
13.12.18 setMessageIdToDequeue() 71
13.12.19 setNull() 71
13.12.20 setPositionOfMessage() 71
13.12.21 setQueueName() 71
13.12.22 setTransformation() 72
13.12.23 setVisibility() 72
13.12.24 setWaitTime() 72
13.13 Date Class 73
13.13.1 Date() 74
13.13.2 addDays() 75
13.13.3 addMonths() 75
13.13.4 daysBetween() 75
13.13.5 fromBytes() 75
13.13.6 fromText() 76
13.13.7 getDate() 77
13.13.8 getSystemDate() 77
13.13.9 isNull() 77
13.13.10 lastDay() 78
13.13.11 nextDay() 78
13.13.12 operator=() 78
13.13.13 operator==() 78
13.13.14 operator!=() 79
13.13.15 operator>() 79
13.13.16 operator>=() 80
13.13.17 operator<() 80
13.13.18 operator<=() 80
13.13.19 setDate() 81
13.13.20 setNull() 81
13.13.21 toBytes() 81
13.13.22 toText() 81
13.13.23 toZone() 82
13.14 Environment Class 83
13.14.1 createConnection() 84
13.14.2 createConnectionPool() 86
13.14.3 createEnvironment() 87
13.14.4 createStatelessConnectionPool() 88
13.14.5 enableSubscription() 89
13.14.6 disableSubscription() 89

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.14.7 getCacheMaxSize() 89
13.14.8 getCacheOptSize() 89
13.14.9 getCacheSortedFlush() 90
13.14.10 getCurrentHeapSize() 90
13.14.11 getLDAPAdminContext() 90
13.14.12 getLDAPAuthentication() 90
13.14.13 getLDAPHost() 90
13.14.14 getLDAPPort() 90
13.14.15 getMap() 91
13.14.16 getNLSLanguage() 91
13.14.17 getNLSTerritory() 91
13.14.18 getOCIEnvironment() 91
13.14.19 getXAConnection() 91
13.14.20 getXAEnvironment() 91
13.14.21 releaseXAConnection() 92
13.14.22 releaseXAEnvironment() 92
13.14.23 setCacheMaxSize() 92
13.14.24 setCacheOptSize() 93
13.14.25 setCacheSortedFlush() 93
13.14.26 setLDAPAdminContext() 93
13.14.27 setLDAPAuthentication() 93
13.14.28 setLDAPHostAndPort() 94
13.14.29 setLDAPLoginNameAndPassword() 94
13.14.30 setNLSLanguage() 94
13.14.31 setNLSTerritory() 95
13.14.32 terminateConnection() 95
13.14.33 terminateConnectionPool() 95
13.14.34 terminateEnvironment() 95
13.14.35 terminateStatelessConnectionPool() 96
13.15 IntervalDS Class 96
13.15.1 IntervalDS() 98
13.15.2 fromText() 99
13.15.3 fromUText() 99
13.15.4 getDay() 100
13.15.5 getFracSec() 100
13.15.6 getHour() 100
13.15.7 getMinute() 100
13.15.8 getSecond() 100
13.15.9 isNull() 100
13.15.10 operator*() 101
13.15.11 operator*=() 101
13.15.12 operator=() 101

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.15.13 operator==() 101
13.15.14 operator!=() 102
13.15.15 operator/() 102
13.15.16 operator/=() 102
13.15.17 operator>() 103
13.15.18 operator>=() 103
13.15.19 operator<() 103
13.15.20 operator<=() 104
13.15.21 operator-() 104
13.15.22 operator-=() 104
13.15.23 operator+() 105
13.15.24 operator+=() 105
13.15.25 set() 105
13.15.26 setNull() 106
13.15.27 toText() 106
13.15.28 toUText() 106
13.16 IntervalYM Class 107
13.16.1 IntervalYM() 108
13.16.2 fromText() 109
13.16.3 fromUText() 109
13.16.4 getMonth() 110
13.16.5 getYear() 110
13.16.6 isNull() 110
13.16.7 operator*() 110
13.16.8 operator*=() 111
13.16.9 operator=() 111
13.16.10 operator==() 111
13.16.11 operator!=() 111
13.16.12 operator/() 112
13.16.13 operator/=() 112
13.16.14 operator>() 112
13.16.15 operator>=() 113
13.16.16 operator<() 113
13.16.17 operator<=() 113
13.16.18 operator-() 114
13.16.19 operator-=() 114
13.16.20 operator+() 114
13.16.21 operator+=() 115
13.16.22 set() 115
13.16.23 setNull() 115
13.16.24 toText() 115
13.16.25 toUText() 116

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.17 Listener Class 116
13.17.1 Listener() 116
13.17.2 getAgentList() 117
13.17.3 getTimeOutForListen() 117
13.17.4 listen() 117
13.17.5 setAgentList() 118
13.17.6 setTimeOutForListen() 118
13.18 Map Class 118
13.18.1 put() 118
13.19 Message Class 119
13.19.1 Message() 121
13.19.2 getAnyData() 121
13.19.3 getAttemptsToDequeue() 121
13.19.4 getBytes() 121
13.19.5 getCorrelationId() 122
13.19.6 getDelay() 122
13.19.7 getExceptionQueueName() 122
13.19.8 getExpiration() 122
13.19.9 getMessageEnqueuedTime() 122
13.19.10 getMessageState() 122
13.19.11 getObject() 122
13.19.12 getOriginalMessageId() 123
13.19.13 getPayloadType() 123
13.19.14 getPriority() 123
13.19.15 getSenderId() 123
13.19.16 isNull() 123
13.19.17 operator=() 123
13.19.18 setAnyData() 124
13.19.19 setBytes() 124
13.19.20 setCorrelationId() 124
13.19.21 setDelay() 124
13.19.22 setExceptionQueueName() 125
13.19.23 setExpiration() 125
13.19.24 setNull() 125
13.19.25 setObject() 126
13.19.26 setOriginalMessageId() 126
13.19.27 setPriority() 126
13.19.28 setRecipientList() 126
13.19.29 setSenderId() 127
13.20 MetaData Class 127
13.20.1 MetaData() 135
13.20.2 getAttributeCount() 135

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.20.3 getAttributeId() 135
13.20.4 getAttributeType() 136
13.20.5 getBoolean() 136
13.20.6 getInt() 136
13.20.7 getMetaData() 137
13.20.8 getNumber() 137
13.20.9 getRef() 137
13.20.10 getString() 137
13.20.11 getTimeStamp() 138
13.20.12 getUInt() 138
13.20.13 getUString() 138
13.20.14 getVector() 139
13.20.15 operator=() 139
13.21 NotifyResult Class 139
13.21.1 getConsumerName() 139
13.21.2 getMessage() 140
13.21.3 getMessageId() 140
13.21.4 getPayload() 140
13.21.5 getQueueName() 140
13.22 Number Class 140
13.22.1 Number() 143
13.22.2 abs() 144
13.22.3 arcCos() 144
13.22.4 arcSin() 145
13.22.5 arcTan() 145
13.22.6 arcTan2() 145
13.22.7 ceil() 145
13.22.8 cos() 145
13.22.9 exp() 145
13.22.10 floor() 146
13.22.11 fromBytes() 146
13.22.12 fromText() 146
13.22.13 hypCos() 147
13.22.14 hypSin() 147
13.22.15 hypTan() 147
13.22.16 intPower() 147
13.22.17 isNull() 147
13.22.18 ln() 148
13.22.19 log() 148
13.22.20 operator++() 148
13.22.21 operator--() 148
13.22.22 operator*() 149

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.22.23 operator/() 149
13.22.24 operator%() 149
13.22.25 operator+() 150
13.22.26 operator-() 150
13.22.27 operator-() 150
13.22.28 operator<() 151
13.22.29 operator<=() 151
13.22.30 operator>() 151
13.22.31 operator>=() 152
13.22.32 operator==() 152
13.22.33 operator!=() 152
13.22.34 operator=() 153
13.22.35 operator*=() 153
13.22.36 operator/=() 153
13.22.37 operator%=() 154
13.22.38 operator+=() 154
13.22.39 operator-=() 154
13.22.40 operator char() 154
13.22.41 operator signed char() 155
13.22.42 operator double() 155
13.22.43 operator float() 155
13.22.44 operator int() 155
13.22.45 operator long() 155
13.22.46 operator long double() 155
13.22.47 operator short() 155
13.22.48 operator unsigned char() 156
13.22.49 operator unsigned int() 156
13.22.50 operator unsigned long() 156
13.22.51 operator unsigned short() 156
13.22.52 power() 156
13.22.53 prec() 157
13.22.54 round() 157
13.22.55 setNull() 157
13.22.56 shift() 157
13.22.57 sign() 158
13.22.58 sin() 158
13.22.59 squareroot() 158
13.22.60 tan() 158
13.22.61 toBytes() 158
13.22.62 toText() 158
13.22.63 trunc() 159
13.23 PObject Class 159

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.23.1 PObject() 161
13.23.2 flush() 161
13.23.3 getConnection() 161
13.23.4 getRef() 161
13.23.5 getSQLTypeName() 161
13.23.6 isLocked() 161
13.23.7 isNull() 162
13.23.8 lock() 162
13.23.9 markDelete() 162
13.23.10 markModified() 162
13.23.11 operator=() 162
13.23.12 operator delete() 163
13.23.13 operator new() 163
13.23.14 pin() 164
13.23.15 setNull() 164
13.23.16 unmark() 164
13.23.17 unpin() 165
13.24 Producer Class 165
13.24.1 Producer() 166
13.24.2 getQueueName() 166
13.24.3 getRelativeMessageId() 166
13.24.4 getSequenceDeviation() 167
13.24.5 getTransformation() 167
13.24.6 getVisibility() 167
13.24.7 isNull() 167
13.24.8 operator=() 167
13.24.9 send() 168
13.24.10 setNull() 168
13.24.11 setQueueName() 168
13.24.12 setRelativeMessageId() 168
13.24.13 setSequenceDeviation() 169
13.24.14 setTransformation() 169
13.24.15 setVisibility() 169
13.25 Ref Class 170
13.25.1 Ref() 171
13.25.2 clear() 171
13.25.3 getConnection() 171
13.25.4 isClear() 171
13.25.5 isNull() 172
13.25.6 markDelete() 172
13.25.7 operator->() 172
13.25.8 operator*() 172

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.25.9 operator==() 172
13.25.10 operator!=() 173
13.25.11 operator=() 173
13.25.12 ptr() 173
13.25.13 setLock() 174
13.25.14 setNull() 174
13.25.15 setPrefetch() 174
13.25.16 unmarkDelete() 175
13.26 RefAny Class 175
13.26.1 RefAny() 176
13.26.2 clear() 176
13.26.3 getConnection() 176
13.26.4 isNull() 176
13.26.5 markDelete() 177
13.26.6 operator=() 177
13.26.7 operator==() 177
13.26.8 operator!=() 177
13.26.9 unmarkDelete() 178
13.27 ResultSet Class 178
13.27.1 cancel() 180
13.27.2 closeStream() 180
13.27.3 getBDouble() 181
13.27.4 getBfile() 181
13.27.5 getBFloat() 181
13.27.6 getBlob() 182
13.27.7 getBytes() 182
13.27.8 getCharSet() 182
13.27.9 getCharSetUString() 182
13.27.10 getClob() 183
13.27.11 getColumnListMetaData() 183
13.27.12 getCurrentStreamColumn() 183
13.27.13 getCurrentStreamRow() 183
13.27.14 getCursor() 184
13.27.15 getDatabaseNCHARParam() 184
13.27.16 getDate() 184
13.27.17 getDouble() 185
13.27.18 getFloat() 185
13.27.19 getInt() 185
13.27.20 getIntervalDS() 185
13.27.21 getIntervalYM() 186
13.27.22 getMaxColumnSize() 186
13.27.23 getNumArrayRows() 186

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.27.24 getNumber() 186
13.27.25 getObject() 187
13.27.26 getRef() 187
13.27.27 getRowid() 187
13.27.28 getRowPosition() 187
13.27.29 getStatement() 188
13.27.30 getStream() 188
13.27.31 getString() 188
13.27.32 getTimestamp() 188
13.27.33 getUInt() 189
13.27.34 getUString() 189
13.27.35 getVector() 189
13.27.36 getVectorOfRefs() 192
13.27.37 isNull() 192
13.27.38 isTruncated() 192
13.27.39 next() 193
13.27.40 preTruncationLength() 193
13.27.41 setBinaryStreamMode() 193
13.27.42 setCharacterStreamMode() 194
13.27.43 setCharSet() 194
13.27.44 setCharSetUString() 194
13.27.45 setDatabaseNCHARParam() 195
13.27.46 setDataBuffer() 195
13.27.47 setErrorOnNull() 196
13.27.48 setErrorOnTruncate() 197
13.27.49 setPrefetchMemorySize() 197
13.27.50 setPrefetchRowCount() 197
13.27.51 setMaxColumnSize() 198
13.27.52 status() 198
13.28 SQLException Class 198
13.28.1 SQLException() 199
13.28.2 getErrorCode() 199
13.28.3 getMessage() 199
13.28.4 getNLSMessage() 199
13.28.5 getNLSUStringMessage() 200
13.28.6 getUStringMessage() 200
13.28.7 getXAErrorCode() 200
13.28.8 isRecoverable() 200
13.28.9 setErrorCtx() 201
13.28.10 what() 201
13.29 StatelessConnectionPool Class 201
13.29.1 getAnyTaggedConnection() 202

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.29.2 getAnyTaggedProxyConnection() 203
13.29.3 getBusyConnections() 204
13.29.4 getBusyOption() 205
13.29.5 getConnection() 205
13.29.6 getIncrConnections() 206
13.29.7 getMaxConnections() 206
13.29.8 getMinConnections() 207
13.29.9 getOpenConnections() 207
13.29.10 getPoolName() 207
13.29.11 getProxyConnection() 207
13.29.12 getStmtCacheSize() 209
13.29.13 getTimeOut() 209
13.29.14 releaseConnection() 209
13.29.15 setBusyOption() 209
13.29.16 setPoolSize() 210
13.29.17 setTimeOut() 210
13.29.18 setStmtCacheSize() 211
13.29.19 terminateConnection() 211
13.30 Statement Class 211
13.30.1 addIteration() 215
13.30.2 closeResultSet() 215
13.30.3 closeStream() 216
13.30.4 disableCaching() 216
13.30.5 execute() 216
13.30.6 executeArrayUpdate() 216
13.30.7 executeQuery() 217
13.30.8 executeUpdate() 217
13.30.9 getAutoCommit() 218
13.30.10 getBatchErrorMode() 218
13.30.11 getBDouble() 218
13.30.12 getBfile() 218
13.30.13 getBFloat() 219
13.30.14 getBlob() 219
13.30.15 getBoolean() 219
13.30.16 getBytes() 221
13.30.17 getCharSet() 221
13.30.18 getCharSetUString() 221
13.30.19 getClob() 222
13.30.20 getConnection() 222
13.30.21 getCurrentIteration() 222
13.30.22 getCurrentStreamIteration() 222
13.30.23 getCurrentStreamParam() 223

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.30.24 getCursor() 223
13.30.25 getDatabaseNCHARParam() 223
13.30.26 getDate() 223
13.30.27 getDMLRowCounts() 224
13.30.28 getDouble() 224
13.30.29 getFloat() 224
13.30.30 getInt() 224
13.30.31 getIntervalDS() 225
13.30.32 getIntervalYM() 225
13.30.33 getMaxIterations() 225
13.30.34 getMaxParamSize() 225
13.30.35 getNumber() 226
13.30.36 getObject() 226
13.30.37 getOCIStatement() 226
13.30.38 getRef() 226
13.30.39 getResultSet() 227
13.30.40 getRowCountsOption() 227
13.30.41 getRowid() 227
13.30.42 getSQL() 227
13.30.43 getSQLUString() 227
13.30.44 getStream() 228
13.30.45 getString() 228
13.30.46 getTimestamp() 228
13.30.47 getUb8RowCount() 228
13.30.48 getUInt() 229
13.30.49 getUpdateCount() 229
13.30.50 getUString() 229
13.30.51 getVector() 229
13.30.52 getVectorOfRefs() 232
13.30.53 isNull() 232
13.30.54 isTruncated() 232
13.30.55 preTruncationLength() 233
13.30.56 registerOutParam() 233
13.30.57 setAutoCommit() 234
13.30.58 setBatchErrorMode() 234
13.30.59 setBDouble() 235
13.30.60 setBfile() 235
13.30.61 setBFloat() 235
13.30.62 setBinaryStreamMode() 236
13.30.63 setBlob() 236
13.30.64 setBoolean() 237
13.30.65 setBytes() 237

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.30.66 setCharacterStreamMode() 237
13.30.67 setCharSet() 238
13.30.68 setCharSetUString() 238
13.30.69 setClob() 239
13.30.70 setDate() 239
13.30.71 setDatabaseNCHARParam() 239
13.30.72 setDataBuffer() 240
13.30.73 setDataBufferArray() 241
13.30.74 setDouble() 242
13.30.75 setErrorOnNull() 243
13.30.76 setErrorOnTruncate() 243
13.30.77 setFloat() 243
13.30.78 setInt() 244
13.30.79 setIntervalDS() 244
13.30.80 setIntervalYM() 244
13.30.81 setMaxIterations() 245
13.30.82 setMaxParamSize() 245
13.30.83 setNull() 246
13.30.84 setNumber() 246
13.30.85 setObject() 247
13.30.86 setPrefetchMemorySize() 247
13.30.87 setPrefetchRowCount() 247
13.30.88 setRef() 248
13.30.89 setRowCountsOption() 248
13.30.90 setRowid() 249
13.30.91 setSQL() 249
13.30.92 setSQLUString() 249
13.30.93 setString() 250
13.30.94 setTimestamp() 250
13.30.95 setUInt() 250
13.30.96 setUString() 251
13.30.97 setVector() 251
13.30.98 setVectorOfRefs() 258
13.30.99 status() 260
13.31 Stream Class 260
13.31.1 readBuffer() 260
13.31.2 readLastBuffer() 261
13.31.3 writeBuffer() 261
13.31.4 writeLastBuffer() 261
13.31.5 status() 262
13.32 Subscription Class 262
13.32.1 Subscription() 263

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.32.2 getCallbackContext() 264
13.32.3 getDatabaseServersCount() 264
13.32.4 getDatabaseServerNames() 264
13.32.5 getNotifyCallback() 264
13.32.6 getPayload() 265
13.32.7 getSubscriptionName() 265
13.32.8 getSubscriptionNamespace() 265
13.32.9 getRecipientName() 265
13.32.10 getPresentation() 265
13.32.11 getProtocol() 266
13.32.12 isNull() 266
13.32.13 operator=() 266
13.32.14 setCallbackContext() 266
13.32.15 setDatabaseServerNames() 266
13.32.16 setNotifyCallback() 267
13.32.17 setNull() 267
13.32.18 setPayload() 267
13.32.19 setPresentation() 267
13.32.20 setProtocol() 268
13.32.21 setSubscriptionName() 268
13.32.22 setSubscriptionNamespace() 268
13.32.23 setRecipientName() 269
13.33 Timestamp Class 269
13.33.1 Timestamp() 271
13.33.2 fromText() 273
13.33.3 getDate() 274
13.33.4 getTime() 274
13.33.5 getTimeZoneOffset() 275
13.33.6 intervalAdd() 275
13.33.7 intervalSub() 276
13.33.8 isNull() 276
13.33.9 operator=() 276
13.33.10 operator==() 276
13.33.11 operator!=() 277
13.33.12 operator>() 277
13.33.13 operator>=() 277
13.33.14 operator<() 278
13.33.15 operator<=() 278
13.33.16 setDate() 278
13.33.17 setNull() 279
13.33.18 setTime() 279
13.33.19 setTimeZoneOffset() 279

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.33.20 subDS() 280
13.33.21 subYM() 280
13.33.22 toText() 280

14 Vector Support in Oracle AI Database

14.1 Oracle Database Vector Support in OCCI 1
14.1.1 Binding Vectors in a Statement Handle 1
14.1.1.1 setInt8Vector() 2
14.1.1.2 setFloatVector() 2
14.1.1.3 setDoubleVector() 2
14.1.1.4 setBinaryVector() 3
14.1.2 PL/SQL Out-Binds for Vectors 3
14.1.2.1 getInt8Vector() 3
14.1.2.2 getFloatVector() 4
14.1.2.3 getDoubleVector() 4
14.1.2.4 getBinaryVector() 4
14.1.3 Defining Vectors from a ResultSet of a Query 5
14.1.3.1 getInt8Vector() 6
14.1.3.2 getFloatVector() 6
14.1.3.3 getDoubleVector() 6
14.1.3.4 getBinaryVector() 7
14.1.3.5 getString() 7
14.1.4 MetaData of Vectors 8
14.1.4.1 Example 8
14.2 Sparse Vectors 11
14.2.1 SparseVector Constructors 12
14.2.1.1 SparseVector() 12
14.2.1.2 SparseVector(uint32_t size,vector<uint32_t> indices,vector<T> values) 12
14.2.1.3 SparseVec(uint32_t size,pair<vector<uint32_t>,vector<T>> &vecp) Ref constructor 13
14.2.2 Binding Sparse Vectors in a Statement Handle 14
14.2.2.1 setInt8Vector() 14
14.2.2.2 setFloatVector() 14
14.2.2.3 setDoubleVector() 15
14.2.2.4 setBinaryVector() 15
14.2.3 PL/SQL out-binds for Sparse Vectors 15
14.2.3.1 getInt8Vector() 16
14.2.3.2 getFloatVector() 16
14.2.3.3 getDoubleVector() 16
14.2.4 Defining Sparse Vectors from a ResultSet of a Query 17
14.2.4.1 getInt8Vector() 17

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.2.4.2 getFloatVector() 17
14.2.4.3 getDoubleVector() 18
14.2.4.4 getString() 18
14.3 Utility Function for SparseVector 19
14.3.1 toDenseVec() 19

Index

Copyright © 1999, 2025, Oracle and/or its affiliates.

List of Examples

1-1	Creating a Table	5
1-2	Specifying Access to a Table	5
1-3	Creating an Object Table	5
1-4	Inserting Data Through Input Bind Variables	6
1-5	Inserting Objects into the Oracle AI Database	6
1-6	Using the Simple SELECT Statement	6
1-7	Using the SELECT Statement with Input Variables	6
1-8	Using PL/SQL to Obtain an Output Variable	7
1-9	Using PL/SQL to Insert Partial Records into Placeholders	7
1-10	Using SQL to Extract Partial Records	8
2-1	How to Determine the Major Client Version and Set Performance Features	1
3-1	How to Create an OCCI Environment	2
3-2	How to Terminate an OCCI Environment	2
3-3	How to Use Environment Scope with Blob Objects	2
3-4	How to Create an Environment and then a Connection to the Database	3
3-5	How to Terminate a Connection to the Database and the Environment	3
3-6	The createConnectionPool() Method	4
3-7	How to Create a Connection Pool	5
3-8	The createProxyConnection() Method	5
3-9	How to Use a StatelessConnectionPool	6
3-10	How to Create and Use a Homogeneous Stateless Connection Pool	7
3-11	How to Create and Use a Heterogeneous Stateless Connection Pool	8
3-12	How to Administer the Database Resident Connection Pools	10
3-13	How to Get a Connection from a Database Resident Connection Pool	11
3-14	Using Client-Side Pool and Server-Side Pool	11
3-15	How to Create a Statement	12
3-16	How to Create a Database Table Using the executeUpdate() Method	13
3-17	How to Add Records Using the executeUpdate() Method	13
3-18	How to Specify a SQL Statement Using the setSQL() Method	13
3-19	How to Reset a SQL Statement Using the setSQL() Method	13
3-20	How to Terminate a Statement Using the terminateStatement() Method	14
3-21	How to Use setxxx() Methods to Set Individual Column Values	15
3-22	How to Specify the IN Parameters of a PL/SQL Stored Procedure	15
3-23	How to Specify OUT Parameters of a PL/SQL Stored Procedure	15
3-24	How to Bind Data in a Streaming Mode	17
3-25	How to Fetch Data in a Streaming Mode Using PL/SQL	18

Copyright © 1999, 2025, Oracle and/or its affiliates.

3-26	How to Read and Write with Multiple Streams	18
3-27	How to Execute an Iterative Operation	20
3-28	How to Fetch Data in Streaming Mode Using ResultSet	21
3-29	SELECT with getUb8RowCount(); simple	25
3-30	SELECT with getUb8RowCount(); with prefetch	25
3-31	SELECT with getUb8RowCount(); array fetch with prefetch	26
3-32	INSERT with getUb8RowCount(); simple	26
3-33	INSERT with getUb8RowCount(); with iterations	26
3-34	UPDATE with getUb8RowCount()	27
3-35	Statement Caching without Connection Pooling	27
3-36	Statement Caching with Connection Pooling	28
4-1	Creating Standalone Objects	3
4-2	Creating Embedded Objects	3
4-3	Two Methods for Operator new() in the Object Type Translator Utility	3
4-4	How to Dynamically Create a Transient Object	4
4-5	How to Create a Transient Object as a Local Variable	4
4-6	How to Create a Persistent Object	4
4-7	How to Create a Transient Object	4
4-8	How to Declare a Custom Type in the Database	5
4-9	How to Create a VARRAY Collection	17
4-10	OTT Support Inheritance	21
4-11	Listing of demo2.sql for a Sample OCCI Application	22
4-12	Listing of demo2.typ for a Sample OCCI Application	22
4-13	Listing of OTT Command that Generates Files for a Sample OCCI Application	22
4-14	Listing of mappings.h for a Sample OCCI Application	23
4-15	Listing of mappings.cpp for a Sample OCCI Application	23
4-16	Listing of demo2.h for a Sample OCCI Application	23
4-17	Listing of demo2.cpp for a Sample OCCI Application	26
4-18	Listing of myDemo.h for a Sample OCCI Application	35
4-19	Listing for myDemo.cpp for a Sample OCCI Application	35
4-20	Listing of main.cpp for a Sample OCCI Application	37
5-1	Definition of the BDOUBLE Data Type	9
5-2	Definition of the BFLOAT Data Type	10
6-1	How to use Identity Column Metadata	2
6-2	How to Obtain Metadata About Attributes of a Simple Database Table	4
6-3	How to Obtain Metadata from a Column Containing User-Defined Types	5
6-4	How to Obtain Object Metadata from a Reference	6

Copyright © 1999, 2025, Oracle and/or its affiliates.

6-5	How to Obtain Metadata About a Select List from a ResultSet Object	7
6-6	How to Obtain Domain Information by getMetaData() on Table Name	7
6-7	How to Obtain Domain Information by getColumnListMetaData() on resultSet Object.	7
7-1	How to Read Non-Streamed BLOBs	4
7-2	How to Read Non-Streamed BFILESs	4
7-3	How to Read Streamed BLOBs	5
7-4	How to Write Non-Streamed BLOBs	5
7-5	How to Write Streamed BLOBs	6
7-6	Updating a CLOB Value	7
7-7	Updating a BFILE Value	7
7-8	How to Use a Persistent Object with a BLOB Attribute	8
7-9	How to Use a Persistent Object with a BFILE Attribute	9
8-1	How to Use the OTT Utility	2
8-2	Object Creation Statements of the OTT Utility	2
8-3	How to Invoke the OTT Utility to Generate C++ Classes	4
8-4	How to use the SCHEMA_NAMES Parameter in OTT Utility	9
8-5	How to Define a Schema for Unicode Support in OTT	10
8-6	How to Use UNICODE=ALL Parameter in OTT	11
8-7	How to Use UNICODE=ONLYCHAR Parameter in OTT	11
8-8	How to Create a User Defined INTYPE File Using the OTT Utility	14
8-9	Listing of ott95a.h	17
8-10	Listing of ott95b.h	18
8-11	How to Represent Object Attributes Using the OTT Utility	20
8-12	How to Map Object Data Types Using the OTT Utility	21
8-13	OUTTYPE File Generated by the OTT Utility	24
8-14	How to Generate C++ Classes Using the OTT Utility	27
8-15	How to Extend C++ Classes Using the OTT Utility	28
9-1	How to Use Globalization and Unicode Support	1
9-2	Using wstring Data Type	2
9-3	Binding UTF8 Data Using the string Data Type	3
9-4	Binding UTF16 Data Using the UString Data Type	3
9-5	Using CLOB and NCLOB Data Types	3
10-1	Creating an Agent	3
10-2	Setting the Agent on the Consumer	3
10-3	Creating an AnyData Message with a String Payload	5
10-4	Determining the Type of the Payload in an AnyData Message	5
10-5	Creating an User-defined Payload	5

Copyright © 1999, 2025, Oracle and/or its affiliates.

10-6	Specifying the Correlation identifier	5
10-7	Specifying the Sender identifier	5
10-8	Specifying the Delay and Expiration times of the message	6
10-9	Specifying message recipients	6
10-10	Specifying the Priority of a Message	6
10-11	Creating a Producer, Setting Visibility, and Enqueuing the Message	6
10-12	Creating a Consumer, Naming the Consumer, and Receiving a Message	7
10-13	Receiving a Message	7
10-14	Specifying dequeuing options	8
10-15	Listening for messages	8
10-16	How to Register for Notifications; Direct Registration	9
10-17	How to Use Open Registration with LDAP	11
11-1	How to Use Transaction Managers with XA	1
12-1	How to Enable TAF for Connection Pooling	3
12-2	How to Insert Records Using the addIteration() method	10
12-3	How to Insert Records Using the executeArrayUpdate() Method	10
12-4	How to use Array Fetch with a ResultSet	11
12-5	How to Modify Rows Iteratively and Handle Errors	11
12-6	How to Use ADRCI for OCCI Application Incidents	17
12-7	How to Use ADRCI for Instant Client	18
12-8	How to Enable and Use the Client Result Cache	19
12-9	Standalone Connection	20
12-10	Session Pool	21
13-1	Converting From an SQL Pre-Defined Type To AnyData Type	8
13-2	Creating an SQL Pre-Defined Type From AnyData Type	8
13-3	Converting From a User-Defined Type To AnyData Type	9
13-4	Converting From a User-Defined Type To AnyData Type	9
13-5	How to Get a Date from Database and Use it in Standalone Calculations	73
13-6	How to Use an Empty IntervalDS Object through Direct Assignment	96
13-7	How to Use an Empty IntervalDS Object Through *Text() Methods	97
13-8	How to Use an Empty IntervalYM Object Through Direct Assignment	107
13-9	How to Use an IntervalYM Object Through ResultSet and toText() Method	107
13-10	How to Retrieve and Use a Number Object	140
13-11	Using Default Timestamp Constructor	272
13-12	Using fromText() method to Initialize a NULL Timestamp Instance	273
13-13	Comparing Timestamps Stored in the Database	273

Copyright © 1999, 2025, Oracle and/or its affiliates.

List of Figures

1-1 The OCCI Development Process 3

4-1 Basic Object Operational Flow 7

8-1 The OTT Utility with OCCI 26

Copyright © 1999, 2025, Oracle and/or its affiliates.

List of Tables

3-1	Normal Data - Not Null and Not Truncated	31
3-2	Null Data	31
3-3	Truncated Data	31
5-1	Summary of Oracle Internal Data Types	2
5-2	External Data Types and Corresponding C++ and OCCI Types	5
5-3	Format of the DATE Data Type	10
5-4	VARNUM Examples	18
5-5	Data Conversions Between External and Internal Data Types	20
5-6	Data Conversions for LOBs	21
5-7	Data Conversions for Date, Timestamp, and Interval Data Types	22
6-1	Attribute Groupings	3
6-2	Attributes that Belong to All Elements	8
6-3	Attributes that Belong to Tables or Views	9
6-4	Attributes Specific to Tables	10
6-5	Attributes that Belong to Procedures or Functions	10
6-6	Attributes that Belong to Package Subprograms	10
6-7	Attributes that Belong to Packages	10
6-8	Attributes that Belong to Types	11
6-9	Attributes that Belong to Type Attributes	12
6-10	Attributes that Belong to Type Methods	13
6-11	Attributes that Belong to Collection Types	14
6-12	Attributes that Belong to Synonyms	15
6-13	Attributes that Belong to Sequences	15
6-14	Attributes that Belong to Columns of Tables or Views	16
6-15	Attributes that Belong to Arguments / Results	16
6-16	Values for ATTR_LIST_TYPE	18
6-17	Attributes Specific to Schemas	18
6-18	Attributes Specific to Databases	18
7-1	Values of Type LobOptionType	10
7-2	Values of Type LobOptionValue	11
8-1	Summary of OTT Utility Parameters	5
8-2	C++ Object Data Type Mappings for Object Type Attributes	19
10-1	Notification Result Attributes; ANONYMOUS and AQ Namespace	12
13-1	Summary of OCCI Classes	1
13-2	Enumerated Values Used by All OCCI Classes	5
13-3	Summary of Agent Methods	5

Copyright © 1999, 2025, Oracle and/or its affiliates.

13-4 OCCI Data Types supported by AnyData Class 9
13-5 Summary of AnyData Methods 10
13-6 Summary of BatchSQLException Methods 17
13-7 Summary of Bfile Methods 18
13-8 Summary of Blob Methods 25
13-9 Summary of Bytes Methods 33
13-10 Summary of Clob Methods 36
13-11 Enumerated Values Used by Connection Class 47
13-12 Summary of Connection Methods 47
13-13 Summary of ConnectionPool Methods 60
13-14 Enumerated Values Used by Consumer Class 65
13-15 Summary of Consumer Methods 66
13-16 Summary of Date Methods 73
13-17 Enumerated Values Used by Environment Class 83
13-18 Summary of Environment Methods 83
13-19 Fields of IntervalDS Class 96
13-20 Summary of IntervalDS Methods 97
13-21 Fields of IntervalYM Class 107
13-22 Summary of IntervalYM Methods 108
13-23 Summary of Listener Methods 116
13-24 Summary of Map Methods 118
13-25 Enumerated Values Used by Message Class 119
13-26 Summary of Message Methods 120
13-27 Enumerated Values Used by MetaData Class 128
13-28 Summary of MetaData Methods 134
13-29 Summary of NotifyResult Methods 139
13-30 Summary of Number Methods 141
13-31 Enumerated Values Used by PObject Class 160
13-32 Summary of PObject Methods 160
13-33 Enumerated Values Used by Producer Class 165
13-34 Summary of Producer Methods 165
13-35 Enumerated Values Used by Ref Class 170
13-36 Summary of Ref Methods 170
13-37 Summary of RefAny Methods 175
13-38 Enumerated Values Used by ResultSet Class 178
13-39 Summary of ResultSet Methods 178
13-40 Summary of SQLException 198

Copyright © 1999, 2025, Oracle and/or its affiliates.

13-41 Enumerated Values Used by StatelessConnectionPool Class 201
13-42 Summary of StatelessConnectionPool Methods 202
13-43 Enumerated Values used by the Statement Class 212
13-44 Statement Methods 212
13-45 Enumerated Values Used by Stream Class 260
13-46 Summary of Stream Methods 260
13-47 Enumerated Values Used by Subscription Class 262
13-48 Summary of Subscription Methods 262
13-49 Fields of Timestamp and Their Legal Ranges 269
13-50 Summary of Timestamp Methods 270

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 AI Database servers. OCCI gives your programs the ability to perform the full range of database operations that are possible with an Oracle AI Database server, including SQL statement processing and object manipulation.

Audience

The *Oracle C++ Call Interface Developer'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++.

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 Oracle Technology Network
- Discussion forum for all Oracle C++ Call Interface related information is at Community — Get Started
- Demos at `$ORACLE_HOME/rdbms/demo`
- *Oracle AI Database Concepts*
- *Oracle AI Database SQL Language Reference*
- *Oracle AI Database Object-Relational Developer's Guide*
- *Oracle AI Database SecureFiles and Large Objects Developer's Guide*
- *Oracle AI Database New Features*
- *Oracle Call Interface Developer's Guide*
- *Oracle AI Database Administrator’s Guide*
- *Oracle AI Database Transactional Event Queues and Advanced Queuing User's Guide*
- *Oracle AI Database Globalization Support Guide*

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Copyright © 1999, 2025, Oracle and/or its affiliates.

1

Introduction to OCCI

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

1.1 Changes in This Release for Oracle C++ Call Interface Programmer's Guide

This section contains the changes in this book for Oracle AI Database 26ai.

New features

- **Oracle C++ Call Interface (OCCI) Support for SQL BOOLEAN Data Type**: Starting with Oracle AI Database 26ai, a new external data type `OCCIBOOL` is introduced to represent BOOLEAN type.

  > ⓘ **See Also**
  > - OCCIBOOL
  > - getBoolean()
  > - setBoolean()

- **Data Use Case Domain Metadata Support in OCCI** :

  > ⓘ **See Also**
  > - getString()
  > - Updated with examples ifor this feature in section Using Metadata (Code Examples).

- **Support for Centralized Configuration Store**: Centralized Configuration Store can be Azure-based or Oracle Cloud Infrastructure based. Centralized Configuration Store manages all application configurations centrally.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Support for Centralized Configuration Store

- **Oracle Database Vector Support in OCCI**: Starting with Oracle AI Database 26ai, support for Vector Data Type is introduced.

> **See Also**
>
> Vector Support in Oracle AI Database

1.2 Overview of OCCI

Oracle C++ Call Interface (OCCI) is an Application Programming Interface (API) that provides C++ applications access to data in an Oracle AI Database. OCCI enables C++ programmers to use the full range of Oracle AI 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 AI Database objects, including client-side access to Oracle AI 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:

- About Benefits of OCCI
- About Building a C++ Application with OCCI
- About Functionality of OCCI
- About Procedural and Nonprocedural Elements

1.2.1 About Benefits of OCCI

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

- Leverages C++ and the Object Oriented Programming paradigm
- Is easy to use

Copyright © 1999, 2025, Oracle and/or its affiliates.

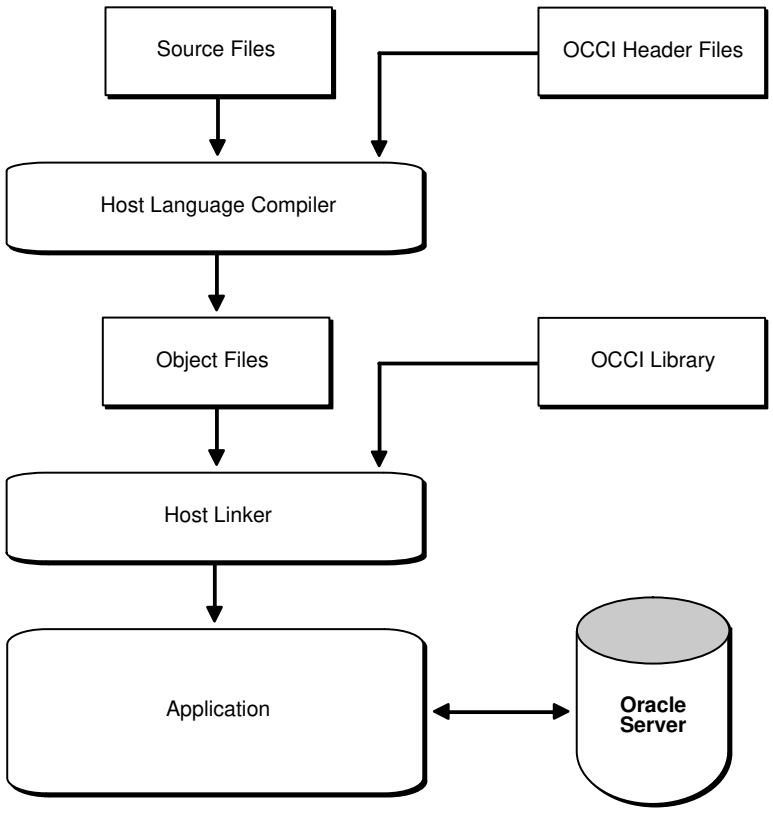
- Is easy to learn for those familiar with JDBC
- Has a navigational interface to manipulate database objects of user-defined types as C++ class instances

Note that OCCI does not support nonblocking mode.

1.2.2 About Building a C++ Application with OCCI

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.

Figure 1-1 The OCCI Development Process



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 AI Database Installation Guide* for more information about compiling and linking an OCCI application for your specific platform

Copyright © 1999, 2025, Oracle and/or its affiliates.

1.2.3 About 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 AI 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

1.2.4 About Procedural and Nonprocedural Elements

Oracle C++ Call Interface (OCCI) enables you to develop scalable, multithreaded applications on multitiered 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 AI 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 AI Database server. For example, an OCCI program can run a query against an Oracle AI 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

written in SQL alone. OCCI also provides facilities for accessing and manipulating objects in an Oracle AI Database server.

1.3 About 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:

- About Data Definition Language Statements
- About Control Statements
- About Data Manipulation Language Statements
- About Queries

1.3.1 About 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.

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

Example 1-1 Creating a Table

```
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

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

Example 1-3 Creating an Object Table

```
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
```

1.3.2 About Control Statements

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

1.3.3 About 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.

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

Example 1-4 Inserting Data Through Input Bind Variables

```
INSERT INTO departments VALUES(:1,:2,:3)
```

Example 1-5 Inserting Objects into the Oracle AI Database

```
INSERT INTO person_info_table
VALUES (person_info_type('450987','123456789','3','146 Winfield Street'))
```

1.3.4 About 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:

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

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

Example 1-6 Using the Simple SELECT Statement

```
SELECT department_name FROM departments
   WHERE department_id = 30
```

Example 1-7 Using the SELECT Statement with Input Variables

```
SELECT first_name, last_name
   FROM employees
   WHERE employee_id = :emp_id
```

1.4 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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 AI 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:

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.

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.

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

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

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

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

1.5 About Special OCCI/SQL Terms

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

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`

Copyright © 1999, 2025, Oracle and/or its affiliates.

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**.

Example 1-10 Using SQL to Extract Partial Records

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

1.6 About 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.

> **See Also**
>
> - *Oracle AI Database Concepts*
> - *Oracle AI Database Object-Relational Developer's Guide*for a more detailed explanation of object types and objects

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

- About Client-Side Object Cache
- About Run-time Environment for Objects
- About Associative and Navigational Interfaces
- About Interoperability with C (OCI)
- About the Metadata Class
- About the Object Type Translator Utility

1.6.1 About 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

1.6.2 About 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 AI 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

1.6.3 About Associative and Navigational Interfaces

Applications that use OCCI can access objects in the database through several types of interfaces, such as SQL `SELECT`, `INSERT`, and `UPDATE` statements, and C++ pointers and references that 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 (`->`) operator on an object reference to transparently materialize the object from the database into the application space.

Copyright © 1999, 2025, Oracle and/or its affiliates.

1.6.4 About Interoperability with C (OCI)

The OCCI application can retrieve the underlying OCI handles and descriptors by calling `getOCI`*`xxx`*`()` 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.

1.6.5 About the 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.

1.6.6 About the 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 AI 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 AI 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 AI 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

1.7 About Additional Support

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

- Building OCCI Demos
- About OCCI on the Oracle Technology Network

1.7.1 Building OCCI Demos

You must install the demonstration programs as described in *Oracle AI Database Examples Installation Guide*. Parts of these demos 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:

     ```
     make -f demo_rdbms.mk occidemos
     ```

   - To make only one demo, use parameters:

     ```
     make -f demo_rdbms.mk demoname
     ```

   - To make a single demo with objects, use parameters:

     ```
     make -f demo_rdbms.mk buildocci EXE=demoname OBJS=demoname.o
     ```

   - To make a single demo with static libraries, use parameters:

     ```
     make -f demo_rdbms.mk buildocci_static EXE=demoname OBJS=demoname.o
     ```

   - For more options, examine the `demo_rdbms.mk` file.

1.7.2 About OCCI on the Oracle Technology Network

You can find additional information on OCCI, including a forum, downloads, and white papers, at: Oracle Technology Network — Oracle C_++_Call Interface.

Copyright © 1999, 2025, Oracle and/or its affiliates.

2

Installation and Upgrading

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

This chapter contains these topics:

- About Installing Oracle C++ Call Interface
- About Upgrading Considerations
- About Determining the Oracle Database Versions
- About Using OCCI with Microsoft Visual C++

2.1 About Installing Oracle C++ Call Interface

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

2.2 About 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.

2.3 About 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.

This sections includes the following topics:

- Determining the Oracle Client Version During Compilation
- About Determining the Oracle Client and Server Versions at Run Time

2.3.1 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

```
#if (OCCI_MAJOR_VERSION > 9)
     env->setCacheSortedFlush(true);  // benefit of performance, if available
#endif
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

2.3.2 About 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.

2.4 Oracle Instant Client and Oracle Instant client Basic Light

Oracle Instant Client enables applications to connect to a local or remote Oracle AI Database for development and production deployment

- **About Oracle Instant Client**: The Oracle Instant Client libraries provide the necessary network connectivity, as well as Oracle AI Database client-side files to create and run Oracle Call Interface (OCI), OCCI, ODBC, and JDBC OCI applications to make full use of Oracle AI Database.
- **About Oracle Instant Client Basic Light**: Oracle Instant Client Basic Light further reduces the disk space requirements of a client installation.

> **See Also**
>
> Installing Oracle Instant Client

2.5 About Using OCCI with Microsoft Visual C++

The Oracle Database 12*c* Release 1 (12.1) includes OCCI libraries for developing applications with Microsoft Visual C++ version 10.0 (.NET 2010 SP1 10.0), Microsoft Visual C++ version 11.0 (.NET 2012 11.0), Microsoft Visual C++ version 12.0 (.NET 2013 12.0), and Intel 12.1 C compilers with Microsoft Visual Studio 2010 STLs. Microsoft Visual C++ version 8.0 and version 9.0 are no longer supported.

Microsoft Visual C++ version 10.0 libraries are installed in the following default locations:

```
ORACLE_BASE\ORACLE_HOME\bin\oraocci12.dll
ORACLE_BASE\ORACLE_HOME\oci\lib\msvc\oraocci12.lib
```

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

```
ORACLE_BASE\ORACLE_HOME\oci\lib\msvc\vc10
```

Microsoft Visual C++ 2012 OCCI libraries are installed in the following default location:

```
ORACLE_BASE\ORACLE_HOME\oci\lib\msvc\vc11
```

When developing OCCI applications with MSVC++ 2012, ensure that the OCCI libraries are correctly selected from this directory for linking and executing.

Microsoft Visual C++ 2013 OCCI libraries are installed in the following default location:

```
ORACLE_BASE\ORACLE_HOME\oci\lib\msvc\vc12
```

When developing OCCI applications with MSVC++ 2013, ensure that the OCCI libraries are correctly selected from this directory for linking and executing.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

Applications that link to `MSVCRTD.DLL`, a debug version of Microsoft C-Runtime, `/MDd` compiler flag, should link with these specific OCCI libraries: `oraocci12d.lib` and `oraocci12d.dll`.

All Instant Client packages contain the versions of the OCCI DLLs that are compatible with Microsoft Visual C++ version 10.0.

Copyright © 1999, 2025, Oracle and/or its affiliates.

3
Accessing Oracle AI Database Using C++

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:

- About Connecting to a Database
- About Pooling Connections
- About Executing SQL DDL and DML Statements
- About Types of SQL Statements in the OCCI Environment
- About Executing SQL Queries
- About Executing Statements Dynamically
- About Using Larger Row Count and Error Code Range Data Types
- About Committing a Transaction
- Caching Statements
- About Handling Exceptions

3.1 About Connecting to a Database

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

This section includes the following topics:

- Creating and Terminating an Environment
- Opening and Closing a Connection
- About Support for Pluggable Databases

3.1.1 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.

All OCCI objects created with the `create`*`xxx`*`()` 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.

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.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

Example 3-1 How to Create an OCCI Environment

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

Example 3-2 How to Terminate an OCCI Environment

```
Environment::terminateEnvironment(env);
```

Example 3-3 How to Use Environment Scope with Blob Objects

```
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);
```

3.1.2 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 creates an environment instance and then uses it to create a database connection for a database user HR with the appropriate 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

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

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

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

3.1.3 About Support for Pluggable Databases

The multitenant architecture enables an Oracle AI database to contain a portable collection of schemas, schema objects, and nonschema objects that appear to an Oracle client as a separate database. A multitenant container database (CDB) is an Oracle AI Database that includes one or more pluggable databases (PDBs).

> **Note**
>
> A multitenant container database is the only supported architecture in Oracle Database 21c.

OCCI clients can connect to a PDB using a service whose pluggable database property has been set to the relevant PDB.

> **See**
>
> - *Oracle AI Database Administrator's Guide* for more information about PDBs and for more details about configuring the services to connect to various PDBs
> - *Oracle® Call Interface Developer's Guide* for information about restrictions while working with PDBs

3.2 About Pooling Connections

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

- About Using Connection Pools
- Using Stateless Connection Pooling

The primary difference between the two is that `StatelessConnectionPool`s 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.

3.2.1 About 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

This section includes the following topics:

- Creating a Connection Pool
- Creating Proxy Connections

3.2.1.1 Creating a Connection Pool

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

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
- `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.

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.

Example 3-6 The createConnectionPool() Method

```
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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 3-7 How to Create a Connection Pool

```
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);
```

3.2.1.2 Creating 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, demonstrated in Example 3-8.

or

```
ConnectionPool->createProxyConnection(
   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.

Example 3-8 The createProxyConnection() Method

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

3.2.2 Using 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

> **Note**
>
> - 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 pool**s. 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`

Copyright © 1999, 2025, Oracle and/or its affiliates.

- `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.

   ```
   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 aterminateStatelessConnectionPool() call on the `Environment` object.

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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, connectString, 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, connectString, maxCon, minCon, incrCon,
            StatelessConnectionPool::PoolType(
               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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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);
```

3.2.3 About 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 AI 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

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

This section includes the following topics:

- Administrating Database Resident Connection Pools
- Using Database Resident Connection Pools

3.2.3.1 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.

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.

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');
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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();
```

3.2.3.2 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 tags 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

```
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);
conn1->terminateStatement(stmt1);
env->terminateConnection(conn1);
```

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

```
StatelessConnectionPool *scPool;
OCCIConnection *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 coonnections */
 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 */
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
                                        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();
    conn1->terminateStatement(stmt1);
/* Release the connection */
    scPool->releaseConnection (conn1);
...
  env->terminateStatelessConnectionPool (scPool);
```

3.3 About 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.

This section includes the following topics:

- Creating a Statement Object
- Creating a Statement Object that Executes SQL Commands
- Reusing the Statement Object
- Terminating a Statement Object

3.3.1 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();
```

3.3.2 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

This section includes the following topics:

- Creating a Database Table
- Inserting Values into a Database Table

3.3.2.1 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))");
```

3.3.2.2 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.

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.

Example 3-17 How to Add Records Using the executeUpdate() Method

```
stmt->executeUpdate("INSERT INTO shopping_basket
   VALUES('MANGO', 3)");
```

3.3.3 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, as demonstrated in Example 3-18.

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.

By using the setSQL() method, 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.

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

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

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

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


Copyright © 1999, 2025, Oracle and/or its affiliates.

3.3.4 Terminating a Statement Object

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

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

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

3.4 About Types of SQL Statements in the OCCI Environment

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

- About Standard Statements use SQL commands with specified values
- Using Parameterized Statements have parameters, or bind variables
- Using 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.

This section also includes the following topics:

- About Streamed Reads and Writes
- About Modifying Rows Iteratively

3.4.1 About Standard Statements

Both Example 3-16 and Example 3-17 demonstrate **standard statement**s 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)`.

3.4.2 Using 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, as demonstrated in Example 3-18.

You then call the `set`*`xxx`*`()` 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 and setInt() method to input the values of these types into the first and second parameters, and the executeUpdate() method 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. 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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
```

3.4.3 Using 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 `set`*`XXX`*`()` methods of the `Statement` class, as demonstrated in Example 3-22.

However, before calling a stored procedure, you must specify the type and size of any `OUT` parameters by calling the registerOutParam() method, as demonstrated in Example 3-23. For `IN/OUT` parameters, use the `set`*`XXX`*`()` methods to pass in the parameter, and `get`*`XXX`*`()` methods to retrieve the results.

You now execute the statement by calling the procedure:

```
stmt->executeUpdate();           // call the procedure
```

Finally, you obtain the output parameters by calling the relevant `get`*`xxx()`* method:

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

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
```

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
```

This section includes the following topic: Using Callable Statements that Use Array Parameters.

3.4.3.1 Using 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:

```
void setDataBufferArray(
   unsigned int paramIndex,
   void *buffer,
   Type type,
   ub4 arraySize,
   ub4 *arrayLength,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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

3.4.4 About 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.

This section includes the following topics:

- Binding Data in Streaming Mode; SELECT/DML and PL/SQL
- Fetching Data in Streaming Mode: PL/SQL
- About Fetching Data in Streaming Mode: ResultSet

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Working with Multiple Streams

3.4.4.1 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`.

> **Note**
>
> For `setBinaryStreamMode()`, the `size` parameter is limited to 32KB (32,768 bytes).

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

```
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
                                                //UPDATE_COUNT_AVAILABLE

Statement *stmt = conn->createStatement("BEGIN testproc(:1); END;");

//if the argument type to testproc is IN or IN/OUT then pass TRUE to
//setCharacterStreamMode or setBinaryStreamMode
stmt->setBinaryStreamMode(1, 32768, TRUE);
```

3.4.4.2 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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
```

3.4.4.3 About Fetching Data in Streaming Mode: ResultSet

About Executing SQL Queries and Example 3-28 provide an explanation of how to use the streaming interface with result sets.

3.4.4.4 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.

> **See Also**
>
> "About Application-Managed Data Buffering"

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, 32768);
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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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);
}
```

3.4.5 About 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

This section includes the following topics:

- Setting the Maximum Number of Iterations
- Setting the Maximum Parameter Size
- Executing an Iterative Operation

3.4.5.1 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.

3.4.5.2 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

3.4.5.3 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

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.

In order to get the number of rows affected by each iteration in the DML execution in Example 3-27, use setRowCountsOption() to enables the feature, followed by getDMLRowCounts() to return the vector of the number of rows. For the total number of rows affected, you can use the return value of executeUpdate() , or call getUb8RowCount().

Example 3-27 How to Execute an Iterative Operation

```
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
```

3.5 About 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.

This section includes the following topics:

- Using the Result Set
- About Specifying the Query
- About Optimizing Performance by Setting Prefetch Count

3.5.1 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

`Statement` class. This method returns an `ResultSet` object that represents the results of a query.

```
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 `get`*`xxx`*`()` methods are used to retrieve the individual columns of the result set, as shown in the following code example:

```
cout << "The basket has:" << endl;

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 `DATA_AVAILABLE`. 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 `STREAM_DATA_AVAILABLE`, until all the streamed data are read successfully.

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

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

```
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;
}
```

3.5.2 About 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`:

```
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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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

3.5.3 About 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.

3.6 About 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.

```
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:

```
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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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

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

This section includes the following topic: About Statement Status Definitions.

3.6.1 About Statement Status Definitions

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

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

3.6.1.1 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.

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

3.6.1.2 PREPARED

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

```
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:

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

3.6.1.3 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:

```
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:

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


Copyright © 1999, 2025, Oracle and/or its affiliates.

3.6.1.4 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:

```
Statement stmt = conn->createStatement("INSERT INTO demo_tab(id) VALUES(99)");
Statemnt::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:

```
stmt->getUb8RowCount();
```

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 a case, you cannot determine the kind of statement that has been executed from the reported status.

3.6.1.5 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_COUNT_AVAILABLE.

3.6.1.6 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

3.7 About Using Larger Row Count and Error Code Range Data Types

Starting with Oracle Database Release 12*c*, Oracle C++ Call Interface supports larger row count and error code range data types. The method that returns the larger row count is getUb8RowCount() in Statement Class.

This has two benefits:

- Applications running a statement that affects more than UB4MAXVAL rows may now see the precise value for the number of rows affected.
- Oracle AI Database can correctly return newer error codes (above `ORA-65535`) to application clients, starting with Oracle Database Release 12*c*. Older clients receive an informative message that indicates error code overflow.

This section contains the following topics:

- "Using Larger Row Count in SELECT Operations"
- "Using Larger Row Count in INSERT, UPDATE, and DELETE Operations"

3.7.1 Using Larger Row Count in SELECT Operations

Method getUb8RowCount() returns the number of rows processed after executing the `SELECT` statement, as `ub8` type. The examples in this section illustrate how to use getUb8RowCount() in various `SELECT` scenarios.

- In the simplest scenario in Example 3-29, the number of rows affected is the same as the number fetched.
- When the prefetch option is set, as demonstrated by Example 3-30, it includes the number of rows prefetched.
- When using an array fetching mechanism in Example 3-31 by invoking the setDataBuffer() interface, getUb8RowCount() returns the total number of rows fetched into user buffers, independent of prefetch option.

Example 3-29 SELECT with getUb8RowCount(); simple

The number of rows affected is the number of rows already fetched.

```
oraub8 largeRowCount = 0;
Statement *stmt = conn->createStatement("SELECT salary FROM employees");
ResultSet *rs = stmt->executeQuery ();
rs->next();
largeRowCount = stmt->getUb8RowCount();
```

Example 3-30 SELECT with getUb8RowCount(); with prefetch

Here the number of rows affected is the same as those fetched in previous iterations, plus the number of rows prefetched in the `next()` call.

```
oraub8 largeRowCount = 0;
Statement *stmt = conn->createStatement("SELECT salary FROM employees");
stmt -> setPrefetchRowCount(prefetch_count);
ResultSet *rs = stmt->executeQuery ();
rs->next();
largeRowCount = stmt->getUb8RowCount();
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 3-31 SELECT with getUb8RowCount(); array fetch with prefetch

Here number of rows affected, value of `largeRowCount`, is the number of rows fetched into user buffer in previous iterations, plus the number of rows fetched in either `next(max)` or `next()` call. It is independent of the value of prefetch.

```
oraub8 largeRowCount = 0;
Statement *stmt=conn->createStatement("SELECT col1 FROM table1");
int max = 20;
int prefetch_count = 10;
ub2 lengthC1[max];
ub4 c1[max];

for (i = 0; i < max; ++i) {
   c1[i] = 0;
   lengthC1[i] = sizeof (c1[i]);
}

stmt -> setPrefetchRowCount(prefetch_count);
ResultSet *rs = stmt->executeQuery();
rs->setDataBuffer (1, c1, OCCIINT, sizeof (ub4), lengthC1);
rs->next(max);

largeRowCount = stmt->getUb8RowCount();
```

3.7.2 Using Larger Row Count in INSERT, UPDATE, and DELETE Operations

For `INSERT`, `UPDATE`, and `DELETE` statements, method getUb8RowCount() returns the number of rows processed by the most recent statement.

Example 3-32 INSERT with getUb8RowCount(); simple

The value of `largeRowCount` is the number of rows inserted, which is `1`.

```
oraub8 largeRowCount = 0;
Statement *stmt = conn->createStatement("INSERT INTO table1 values (:1)");
stmt->setNumber(1, 100);
stmt->executeUpdate();
largeRowCount = stmt->getUb8RowCount();
```

Example 3-33 INSERT with getUb8RowCount(); with iterations

Here the value of `largeRowCount` is equal to `max`.

```
int max;
oraub8 largeRowCount = 0;
Statement *stmt=conn->createStatement("INSERT INTO table1 values (:1)");
stmt->setMaxIterations (max);

for(i = 0; i < max-1; i++) {
   stmt->setNumber(1, 100);
   stmt->addIteration ();
}

stmt->setNumber(1, 100);
stmt->executeUpdate();
largeRowCount = stmt->getUb8RowCount();
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 3-34 UPDATE with getUb8RowCount()

Here the value of `largeRowCount` is the number of rows updated.

```
oraub8 largeRowCount = 0;
Statement *stmt=conn->createStatement(
   "UPDATE table1 SET COL1 = COL1+100 WHERE COL1=:1");
stmt->setNumber(1, 200);
stmt->executeUpdate();
largeRowCount = stmt->getUb8RowCount();
```

3.8 About 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:

```
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:

```
Statement::setAutoCommit(FALSE);
```

3.9 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 3-35 and Example 3-36 for typical usage scenarios.

Example 3-35 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 3-36 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 cac.hing 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, connecstr,
                                minConn, maxConn, incrConn);
   ```

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

   ```
   Stateless ConnectionPool *conPool = env->createStatelessConnectionPool(
                                       username, password, connecstr,
                                       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);
   ```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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, connecstr);
   ```

   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():

   ```
   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().

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

3.10 About 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

`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:

```
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:

```
catch (SQLException &sqlExcp)
{
   cerr <<sqlExcp.getErrorCode << ": " << sqlExcp.getErrorMessage() << 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" in Optimizing Performance of C++ Applications
- *Oracle AI Database Error Messages Reference*for more information about Oracle error messages.

This section includes the following topic: About Handling Null and Truncated Data.

3.10.1 About Handling Null and Truncated Data

In general, OCCI does not cause an exception when the data value retrieved by using the `get`*`xxx`*`()` 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 `setError`*`xxx`*`()` 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`:

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


Copyright © 1999, 2025, Oracle and/or its affiliates.

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

```
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 3-1 Normal Data - Not Null and Not Truncated

Return Code	Indicator - not provided	Indicator - provided
Not provided	`error = 0`	`error = 0` `indicator = 0`

Provided	`error = 0`
	`return code = 0`
		`error = 0`
		`indicator = 0`
		`return code = 0`

Table 3-2 Null Data

Return Code	Indicator - not provided	Indicator - provided
Not provided	`SQLException` `error = 1405`	`error = 0` `indicator = -1`

Provided	`SQLException`
	`error = 1405`
	`return code = 1405`
		`error = 0`
		`indicator = -1`
		`return code = 1405`

Table 3-3 Truncated Data

Return Code	Indicator - not provided	Indicator - provided

Not provided	`SQLException`
	`error = 1406`
		`SQLException`
		`error = 1406`
		`indicator = data_len`

Provided	`error = 24345`
	`return code = 1405`
		`error = 24345`
		`indicator = data_len`
		`return code = 1406`

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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

4

Object Programming

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

4.1 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 (REFs). 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 REFs to those objects. The application then uses those REFs 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

4.2 About Working with Objects in C++ with 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:

- About Persistent Objects
- About Transient Objects
- About Values

4.2.1 About Persistent Objects

A **persistent object** is an object which is stored in an Oracle AI 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 `REF`s 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

4.2.2 About 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 Example 4-3Example 4-3:

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.

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.

> **See Also**
>
> - *Oracle AI Database Concepts*for more information about objects

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
}
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 4-4 How to Dynamically Create a Transient Object

```
Person *p = new Person();
```

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

```
Person p;
```

4.2.3 About 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.

4.3 About 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

4.3.1 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 4-7 shows hot to create an instance of the transient object `ADDRESS`.

Example 4-6 How to Create a Persistent Object

```
CREATE TYPE ADDRESS AS OBJECT (
   state CHAR(2),
   zip_code CHAR(5));
CREATE TABLE ADDR_TAB of ADDRESS;
ADDRESS *addr = new(conn, "ADDR_TAB") ADDRESS("CA", "94065");
```

Example 4-7 How to Create a Transient Object

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


Copyright © 1999, 2025, Oracle and/or its affiliates.

4.3.2 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.

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 `PObjects`) can be either transient or persistent. The methods `writeSQL()` and `readSQL()` 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**
>
> Object Type Translator Utility for more information about the OTT utility

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

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

The OTT utility produces the following C++ class:

```
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);
}
```

4.4 About Developing a C++ Application using OCCI

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

This section discusses the following topics:

- Developing Basic Object Program Structure
- About Basic Object Operational Flow

Copyright © 1999, 2025, Oracle and/or its affiliates.

4.4.1 Developing 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 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.

See Also

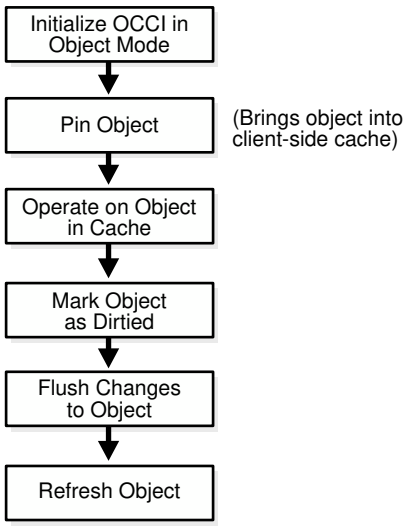
- Accessing Oracle AI Database Using C++ for information about using OCCI to connect to a database server, process SQL statements, and allocate handles
- Object Type Translator Utility for information about the OTT utility
- OCCI Application Programming Interface for descriptions of OCCI relational functions and the `Connect` class and the `getMetaData()` method

Copyright © 1999, 2025, Oracle and/or its affiliates.

4.4.2 About Basic Object Operational Flow

Figure 4-1 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 4-1 are discussed in the following sections:

- About Initializing OCCI in Object Mode
- About Pinning anObject
- About Operating on an Object in Cache
- About Flushing Changes to the Object
- About Deletion of an Object

4.4.2.1 About Initializing 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 ithe `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:

```
Environment *env;
Connection *con;
Statement *stmt;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.

4.4.2.2 About Pinning anObject

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.

4.4.2.3 About Operating on an 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.

4.4.2.4 About Flushing Changes to the Object

Whenever changes are made to object copies in the object cache, your application is responsible for flushing the changed object to the database.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

4.4.2.5 About 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.

4.5 Migrating C++ Applications to Oracle 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 JDeveloper9*i* 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.

4.6 Overview of Associative Access

You can employ SQL within OCCI to retrieve objects, and to perform DML operations.

This section discusses the following topics:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Using SQL to Access Objects
- Inserting and Modifying Values

> **See Also**
>
> Complete code listing of the demonstration programs

4.6.1 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:

```
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.

4.6.2 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`:

```
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();
```

4.7 Overview of Navigational Access

By using navigational access, you engage in a series of operations.

This section discusses the following topics:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Retrieving an Object Reference (REF) from the Database Server
- Pinning an Object
- Manipulating Object Attributes
- About Marking Objects and Flushing Changes
- Marking an Object as Modified (Dirty)
- About Recording Changes in the Database
- Collecting Garbage in the Object Cache
- About Ensuring Transactional Consistency of References

> **See Also**
>
> Complete code listing of the demonstration programs

4.7.1 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 (`REF`s) to one or more objects.

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

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

```
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.

```
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**
>
> "About Executing SQL DDL and DML Statements" for general information about preparing and executing SQL statements

4.7.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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:

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

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

```
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.

4.7.3 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:

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

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

```
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.

4.7.4 About 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.

4.7.5 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).

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

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

4.7.6 About 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, OCCI 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.

4.7.7 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:

```
Maximum cache size = optimal_size + optimal_size * max_size_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;`

"About Pinning anObject" 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 OCCI uses STLs. You should therefore set the cache size to a low value to avoid high memory use before garbage collection activates.

> **See Also**
>
> OCCI Application Programming Interface for details.

4.7.8 About Ensuring Transactional Consistency of References

As described in the previous section, dereferencing a `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 `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 `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.

4.8 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- About Prefetching Complex Objects

> **See Also**
>
> Complete code listing of the demonstration programs

4.8.1 Retrieving Complex Objects

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

```
// prefetch attributes of the specified type name up to the specified depth
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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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:

```
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:

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

4.8.2 About 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 READ or SELECT privilege for all prefetched objects. Objects in the complex object for which the application does not have READ or SELECT privilege cannot prefetched.

An entire vector of Refs 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 of Refs from n to 1, and tracks the newly pinned objects through an OUT parameter vector.

4.9 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

VARRAY and an object that contains an attribute of type VARRAY, and the resulting C++ declaration that OTT generates.

> **See Also**
>
> Complete code listing of the demonstration programs

Example 4-9 How to Create a VARRAY Collection

```
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:

```
class PERSON : public PObject
{
   protected:
      string name;
      vector< Ref< ADDRESS > > addr_1;

   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);
}
```

This section includes the following topics:

- Fetching Embedded Objects
- About Nullness

4.9.1 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.

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


Copyright © 1999, 2025, Oracle and/or its affiliates.

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

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

Your OCCI application could then issue the following SQL statement:

```
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".

4.9.2 About 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 `NULL`s 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.

4.10 About 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.

4.11 About 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

4.12 About 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 AI 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:

```
CREATE TYPE Person_t AS OBJECT
(  ssn NUMBER,
   name VARCAHR2(30),
   address VARCHAR2(100)) NOT FINAL;
```

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));
```

> **See Also**
>
> - "About OTT Support for Type Inheritance" for the classes generated by OTT for this example.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

This section includes the following topics:

- About Substitutability
- Declaring NOT INSTANTIABLE Types and Methods
- About OCCI Support for Type Inheritance
- About OTT Support for Type Inheritance

4.12.1 About 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))
```

4.12.2 Declaring 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.

4.12.3 About OCCI Support for Type Inheritance

The following calls support type inheritance:

- About Connection::getMetaData()
- About Bind and Define Functions

4.12.3.1 About 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.

4.12.3.2 About 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 `get`*`xxx`*`()` methods to fetch the data also support substitutability.

4.12.4 About 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:

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 4-10 OTT Support Inheritance

```
class <typename> : public <parentTypename>
{
   protected:
      <OCCItype1> <attributename1>;
      ...
      <OCCItypen> <attributenamen>;

   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,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
                        unsigned int &typeNameLen) const;
      <typename> (void *ctx) : <parentTypename>(ctx) { };
      static void *readSQL(void *ctx);
      virtual void readSQL(AnyData& stream);
      static void writeSQL(void *obj, void *ctx);
      virtual void writeSQL(AnyData& stream);
}
```

4.13 A Sample OCCI Application

This section describes a sample OCCI application that uses some features discussed in this chapter.

Example 4-11 Listing 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 user name `demousr`; the system prompts for the password.

```
ott userid=demousr intype=demo2.typ code=cpp hfile=demo2.h
   cppfile=demo2.cpp mapfile=mappings.cpp attraccess=private
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
   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);
   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, 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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 > > &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, 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:
      MyPerson(Number id_i, MyFullName *name_i, const Ref<MyAddress>& addr_i) ;
      MyPerson(void *ctxOCCI_);
      void move(const Ref<MyAddress>& new_addr);
      void displayInfo();
      MyPerson();
};
/* changes  end here */

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, 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,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      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 */
#endif
```

Example 4-17 Listing of demo2.cpp for a Sample OCCI Application

```
#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)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
{
  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)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  {
    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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
}

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, table,
            (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_);
  }
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
{
  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, table,
            (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
{
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  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(void *objectOCCI_, void *ctxOCCI_)
{
  CPerson *objOCCI_ = (CPerson *) 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
}

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,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
                                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())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void CStudent::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   CPerson::readSQL(streamOCCI_);
   SCHOOL_NAME = streamOCCI_.getString();
}

void CStudent::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CStudent *objOCCI_ = (CStudent *) 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 CStudent::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   CPerson::writeSQL(streamOCCI_);
   streamOCCI_.setString(SCHOOL_NAME);
}
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
CStudent::~CStudent()
{
  int i;
}
```

Let us assume OTT generates `FULL_NAME`, `ADDRSESS`, `PERSON`, and `PFGRFDENT` 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

```
#ifndef MYDEMO_ORACLE
#define MYDEMO_ORACLE

#include <string>

#ifndef DEMO2_ORACLE
#include <demo2.h>
#endif

using namespace std;
using namespace oracle::occi;

// 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();
};

#endif
```

Example 4-19 Listing for myDemo.cpp for a Sample OCCI Application

```
#ifndef DEMO2_ORACLE
#include <demo2.h>
#endif

using namespace std;

/* initialize MyFullName */
MyFullName::MyFullName(string first_name,string last_name)
{
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
 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 */
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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();
   }
}

MyPerson::MyPerson()
{
}

MyStudent::MyStudent(void *ctxOCCI_) : CStudent(ctxOCCI_)
{
}
```

Example 4-20 Listing of main.cpp for a Sample OCCI Application

```
#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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      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;
}
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

5
Data Types

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

5.1 Overview of Oracle Data Types

Accurate communication between your C++ program and the Oracle AI 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 AI 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.

This section includes the following topic: About OCCI Type and Data Conversion.

5.1.1 About 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 `set`*`xxx`*`()` 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

`ResultSet` object, the external representation of the retrieved data must be specified. This is done by calling a `get`*`xxx`*`()` 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

5.2 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

Internal Data Type	Maximum Size
`BFILE`	4 gigabytes
`BINARY_DOUBLE`	8 bytes
`BINARY_FLOAT`	4 bytes
`CHAR`	2,000 bytes
`DATE`	7 bytes
`INTERVAL DAY TO SECOND REF`	11 bytes
`INTERVAL YEAR TO MONTH REF`	5 bytes
`LONG`	2 gigabytes (2^31-1 bytes)
`LONG RAW`	2 gigabytes (2^31-1 bytes)
`NCHAR`	2,000 bytes
`NUMBER`	21 bytes
`NVARCHAR2`	32,767 bytes

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 5-1 (Cont.) Summary of Oracle Internal Data Types

Internal Data Type	Maximum Size
RAW	2000 bytes (standard), 32,767 bytes (extended)
REF	Not Applicable
BLOB	4 gigabytes
CLOB	4 gigabytes
NCLOB	4 gigabytes
ROWID	10 bytes
TIMESTAMP	11 bytes
TIMESTAMP WITH LOCAL TIME ZONE	7 bytes
TIMESTAMP WITH TIME ZONE	13 bytes
UROWID	4000 bytes
User-defined type (object type, VARRAY, nested table)	Not Applicable
VARCHAR2	4000 bytes (standard), 32,767 bytes extended

See Also

- *Oracle AI Database SQL Language Reference*
- *Oracle AI Database Concepts*

This section includes the following topics:

- Character Strings and Byte Arrays
- Universal Rowid (UROWID)

5.2.1 Character Strings and Byte Arrays

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

(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 AI Database tables.

5.2.2 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_AVC` (`CHARZ`)
- `OCCI_SQLT_VST` (string)
- `OCCI_SQLT_RDD` (`ROWID` descriptor)

5.3 External Data Types

OCCI application communicate with the Oracle AI 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 `OCCI`*`xxx`* (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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- The TIMESTAMP and TIMESTAMP 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 5-2 External Data Types and Corresponding C++ and OCCI Types

External Data Type	C++ Type	OCCI Type	Usage Notes
16 bit signed INTEGER	signed short, signed int	OCCIINT	Use with setDataBuffer(), setDataBufferArray().
32 bit signed INTEGER	signed int, signed long	OCCIINT	Use with setDataBuffer(), setDataBufferArray().
8 bit signed INTEGER	signed char	OCCIINT	Use with setDataBuffer(), setDataBufferArray().
BFILE	Bfile	OCCIBFILE	Use with regiserOutParam().
FBFILE	OCILobLocator	OCCI_SQLT_FILE	Use with setDataBuffer(), setDataBufferArray().
BLOB	OCILobLocator	OCCI_SQLT_BLOB	Use with setDataBuffer(), setDataBufferArray().
BLOB	Blob	OCCIBLOB	Use with regiserOutParam().
BOOL	bool	OCCIBOOL	Use with regiserOutParam().
BYTES	Bytes	OCCIBYTES	Use with regiserOutParam().
CHAR	char[n]	OCCI_SQLT_AFC	Use with setDataBuffer(), setDataBufferArray().
CHAR	string	OCCICHAR	Use with regiserOutParam().
CLOB	OCILobLocator	OCCI_SQLT_CLOB	Use with setDataBuffer(), setDataBufferArray().
CHARZ	char[n+1]	OCCI_SQLT_RDD	Use with setDataBuffer(), setDataBufferArray().
CLOB	Clob	OCCICLOB	Use with regiserOutParam().
CURSOR	ResultSet	OCCICURSOR	Use with regiserOutParam().
DATE	char[7]	OCCI_SQLT_DAT	Use with setDataBuffer(), setDataBufferArray().
DATE	Date	OCCIDATE	Use with regiserOutParam().
DOUBLE	double	OCCIDOUBLE	Use with regiserOutParam().
FLOAT	float, double	OCCIFLOAT	Use with setDataBuffer(), setDataBufferArray().

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 5-2 (Cont.) External Data Types and Corresponding C++ and OCCI Types

External Data Type	C++ Type	OCCI Type	Usage Notes
FLOAT	float	OCCIFLOAT	Use with regiserOutParam().
INT	int	OCCIINT	Use with regiserOutParam().
INTERVAL DAY TO SECOND	char[11]	OCCI_SQLT_INTERVAL_DS	Use with setDataBuffer(), setDataBufferArray().
INTERVAL YEAR TO MONTH	char[5]	OCCI_SQLT_INTERVAL_YM	Use with setDataBuffer(), setDataBufferArray().
INTERVALDS	IntervalDS	OCCIINTERVALDS	Use with regiserOutParam().
INTERVALYM	IntervalYM	OCCIINTERVALYM	Use with regiserOutParam().
LONG	char[n]	OCCI_SQLT_LNG	Use with setDataBuffer(), setDataBufferArray().
LONG RAW	unsigned char[n]	OCCI_SQLT_LBI	Use with setDataBuffer(), setDataBufferArray().
LONG VARCHAR	char[n+siezeof(integer)]	OCCI_SQLT_LVC	Use with setDataBuffer(), setDataBufferArray().
LONG VARRAW	unsigned char[n+siezeof(integer)]	OCCI_SQLT_LVB	Use with setDataBuffer(), setDataBufferArray().
METADATA	MetaData	OCCIMETADATA	Use with regiserOutParam().
NAMED DATA TYPE	struct	OCCI_SQLT_NTY	Use with setDataBuffer(), setDataBufferArray().
NATIVE DOUBLE	double	OCCIBDOUBLE	Use with setDataBuffer(), setDataBufferArray().
NATIVE DOUBLE	Bdouble, double	OCCIBDOUBLE	Use with regiserOutParam().
NATIVE FLOAT	float	OCCIBFLOAT	Use with setDataBuffer(), setDataBufferArray().
NATIVE FLOAT	BFloat, float	OCCIBFLOAT	Use with regiserOutParam().
null terminated STRING	char[n+1]	OCCI_SQLT_STR	Use with setDataBuffer(), setDataBufferArray().
NUMBER	unsigned char[21]	OCCI_SQLT_NUM	Use with setDataBuffer(), setDataBufferArray().
NUMBER	Number	OCCINUMBER	Use with regiserOutParam().
POBJECT	User defined types generated by OTT utility.	OCCIPOBJECT	Use with regiserOutParam().

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 5-2 (Cont.) External Data Types and Corresponding C++ and OCCI Types

External Data Type	C++ Type	OCCI Type	Usage Notes
RAW	unsigned char[n]	OCCI_SQLT_BIN	Use with setDataBuffer(), setDataBufferArray().
REF	OCIRef	OCCI_SQLT_REF	Use with setDataBuffer(), setDataBufferArray().
REF	Ref	OCCIREF	Use with regiserOutParam().
REFANY	RefAny	OCCIREFANY	Use with regiserOutParam().
ROWID	OCIRowid	OCCI_SQLT_RID	Use with setDataBuffer(), setDataBufferArray().
ROWID	Bytes	OCCIROWID	Use with regiserOutParam().
ROWID descriptor	OCIRowid	OCCI_SQLT_RDD	Use with setDataBuffer(), setDataBufferArray().
STRING	STL string	OCCISTRING	Use with regiserOutParam().
TIMESTAMP	char[11]	OCCI_SQLT_TIMESTAMP	Use with setDataBuffer(), setDataBufferArray().
TIMESTAMP	Timestamp	OCCITIMESTAMP	Use with regiserOutParam().
TIMESTAMP WITH LOCAL TIME ZONE	char[7]	OCCI_SQLT_TIMESTAMP_LTZ	Use with setDataBuffer(), setDataBufferArray().
TIMESTAMP WITH TIME ZONE	char[13]	OCCI_SQLT_TIMESTAMP_TZ	Use with setDataBuffer(), setDataBufferArray().
UNSIGNED INT	unsigned int	OCCIUNSIGNED_INT	Use with setDataBuffer(), setDataBufferArray().
UNSIGNED INT	unsigned int	OCCIUNSIGNED_INT	Use with regiserOutParam().
VARCHAR	char[n+sizeof(short integer)]	OCCI_SQLT_VCS	Use with setDataBuffer(), setDataBufferArray().
VARCHAR2	char[n]	OCCI_SQLT_CHR	Use with setDataBuffer(), setDataBufferArray().
VARNUM	char[22]	OCCI_SQLT_VNU	Use with setDataBuffer(), setDataBufferArray().
VARRAW	unsigned char[n+sizeof(short integer)]	OCCI_SQLT_VBI	Use with setDataBuffer(), setDataBufferArray().
VECTOR	STL vector	OCCIVECTOR	Use with regiserOutParam().

This section includes the following topic: Description of External Data Types.

Copyright © 1999, 2025, Oracle and/or its affiliates.

5.3.1 Description of External Data Types

This section provides a description for each of the external data types:

- BFILE
- BDOUBLE
- BFLOAT
- BLOB
- CHAR
- CHARZ
- CLOB
- DATE
- FLOAT
- INTEGER
- INTERVAL DAY TO SECOND
- INTERVAL YEAR TO MONTH
- LONG
- LONG RAW
- LONG VARCHAR
- LONG VARRAW
- NCLOB
- NUMBER
- OCCI BFILE
- OCCI BLOB
- OCCI BYTES
- OCCI CLOB
- OCCI DATE
- OCCI INTERVALDS
- OCCI INTERVALYM
- OCCI NUMBER
- OCCI POBJECT
- OCCI REF
- OCCI REFANY
- OCCI STRING
- OCCI TIMESTAMP
- OCCI VECTOR
- RAW
- REF

Copyright © 1999, 2025, Oracle and/or its affiliates.

- ROWID
- STRING
- TIMESTAMP
- TIMESTAMP WITH LOCAL TIME ZONE
- TIMESTAMP WITH TIME ZONE
- UNSIGNED INT
- VARCHAR
- VARCHAR2
- VARNUM
- VARRAW
- NATIVE DOUBLE
- NATIVE FLOAT

5.3.1.1 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.

5.3.1.2 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

```
struct BDouble
{
  double value;
  bool isNull;

 BDouble()
 {
    isNull = false;
    value = 0.;
 }
};
```

5.3.1.3 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 5-2 Definition of the BFLOAT Data Type

```
struct BFloat
{
  float value;
  bool isNull;

 BFloat()
 {
    isNull = false;
    value = 0.;
 }
};
```

5.3.1.4 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. `BLOB`s 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.

5.3.1.5 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.

5.3.1.6 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.

5.3.1.7 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. `CLOB`s 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.

5.3.1.8 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

Example	Byte 1: Century	Byte 2: Year	Byte 3: Month	Byte 4: Day	Byte 5: Hour	Byte 6: Minute	Byte 7: Second
1: 01-JUN-2000, 3:17PM	120	100	6	1	16	18	1
2: 01-JAN-4712 BCE	53	88	1	1	1	1	1

Copyright © 1999, 2025, Oracle and/or its affiliates.

5.3.1.8.1 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`.

5.3.1.8.2 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.

5.3.1.9 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

5.3.1.10 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.

5.3.1.11 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)

5.3.1.12 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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

5.3.1.13 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.

5.3.1.14 LONG RAW

The external data type LONG RAW is similar to the external data type RAW, except that it stores up to 2 gigabytes.

5.3.1.15 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.

5.3.1.16 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.

5.3.1.17 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.

5.3.1.18 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

5.3.1.19 OCCI BFILE

> **See Also**
>
> OCCI Application Programming Interface, Bfile Class

5.3.1.20 OCCI BLOB

> **See Also**
>
> OCCI Application Programming Interface, Blob Class

5.3.1.21 OCCIBOOL

Starting with Oracle AI Database 26ai, a new external data type OCCIBOOL is introduced to represent BOOLEAN type.

5.3.1.22 OCCI BYTES

> **See Also**
>
> OCCI Application Programming Interface, Bytes Class

Copyright © 1999, 2025, Oracle and/or its affiliates.

5.3.1.23 OCCI CLOB

> **See Also**
>
> OCCI Application Programming Interface, Clob Class

5.3.1.24 OCCI DATE

> **See Also**
>
> OCCI Application Programming Interface, Date Class

5.3.1.25 OCCI INTERVALDS

> **See Also**
>
> OCCI Application Programming Interface, IntervalDS Class

5.3.1.26 OCCI INTERVALYM

> **See Also**
>
> OCCI Application Programming Interface, IntervalYM Class

5.3.1.27 OCCI NUMBER

> **See Also**
>
> OCCI Application Programming Interface, Number Class

5.3.1.28 OCCI POBJECT

> **See Also**
>
> OCCI Application Programming Interface, PObject Class

Copyright © 1999, 2025, Oracle and/or its affiliates.

5.3.1.29 OCCI REF

> **See Also**
>
> OCCI Application Programming Interface, Ref Class

5.3.1.30 OCCI REFANY

> **See Also**
>
> OCCI Application Programming Interface, RefAny Class

5.3.1.31 OCCI STRING

The external data type `OCCI STRING` corresponds to an `STL string`.

5.3.1.32 OCCI TIMESTAMP

> **See Also**
>
> OCCI Application Programming Interface, Timestamp Class

5.3.1.33 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.

5.3.1.34 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. If the `init.ora` parameter `max_string_size = standard` (default value), the maximum length of a `RAW` can be 2000 bytes. If the `init.ora` parameter `max_string_size = extended`, the maximum length of a `RAW` can be 32767 bytes. See the `init.ora` parameter MAX_STRING_SIZE in *Oracle AI Database Reference* for more information about extended data types.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

5.3.1.35 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`.

5.3.1.36 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.

5.3.1.37 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.

5.3.1.38 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`.

5.3.1.39 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

5.3.1.40 TIMESTAMP WITH TIME ZONE

The external data type `TIMESTAMP WITH TIME ZONE` is a variant of `TIMESTAMP` that includes a **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`. For example, you might specify `TIMESTAMP(0) WITH TIME ZONE` as a literal as follows:

```
TIMESTAMP '1997-01-31 09:26:50+02.00'
```

5.3.1.41 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.

5.3.1.42 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.

5.3.1.43 VARCHAR2

The external data type `VARCHAR2` is a variable-length string of characters up to 4000 bytes. If the `init.ora` parameter `max_string_size = standard` (default value), the maximum length of a `VARCHAR2` can be 4000 bytes. If the `init.ora` parameter `max_string_size = extended`, the maximum length of a `VARCHAR2` can be 32767 bytes. See the `init.ora` parameter MAX_STRING_SIZE in *Oracle AI Database Reference* for more information about extended data types.

5.3.1.44 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

Decimal Value	Length Byte	Exponent Byte	Mantissa Bytes	Terminator Byte
0	1	128	NA[1]	NA
5	2	193	6	NA

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 5-4 (Cont.) VARNUM Examples

Decimal Value	Length Byte	Exponent Byte	Mantissa Bytes	Terminator Byte
-5	3	62	96	102
2767	3	194	28, 68	NA
-2767	4	61	74, 34	102
100000	2	195	11	NA
1234567	5	196	2, 24, 46, 68	NA

1 NA means not applicable.

5.3.1.45 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.

5.3.1.46 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.

5.3.1.47 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

5.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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

NA[1]	Internal Data Types									
External Data Types	**VARCHAR2**	**NUMBER**	**LONG**	**ROWID**	**DATE**	**RAW**	**LONG RAW**	**CHAR**	**BFLOAT**	**BDOUBLE**
CHAR	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I[3, 5]	I/O	I/O	I/O
CHARZ	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I[3, 5]	I/O	NA	NA
DATE	I/O	NA	I	NA	I/O	NA	NA	I/O	NA	NA
DECIMAL	I/O[4]	I/O	I	NA	NA	NA	NA	I/O[4]	NA	NA
FLOAT	I/O[4]	I/O	I	NA	NA	NA	NA	I/O[4]	I/O	I/O
INTEGER	I/O[4]	I/O	I	NA	NA	NA	NA	I/O[4]	I/O	I/O
LONG	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3, 5]	I/O	I/O	II/O
LONG RAW	O[6]	NA	I[5, 6]	NA	NA	I/O	I/O	O[6]	NA	NA
LONG VARCHAR	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3, 5]	I/O	I/O	I/O
LONG VARRAW	I/O[6]	NA	I[5, 6]	NA	NA	I/O	I/O	I/O[6]	NA	NA
NUMBER	I/O[4]	I/O	I	NA	NA	NA	NA	I/O[4]	I/O	I/O
OCCI BDouble	I/O	1/O	I	NA	NA	NA	NA	I/O	I/O	I/O
OCCI BFloat	I/O	1/O	I	NA	NA	NA	NA	I/O	I/O	I/O
OCCI Bytes	I/O[6]	NA	I[5, 6]	NA	NA	I/O	I/O	I/O[6]	NA	NA
OCCI Date	I/O[2]	NA	I	NA	I/O	NA	NA	I/O	NA	NA
OCCI Number	I/O[4]	I/O	I	NA	NA	NA	NA	I/O[4]	I/O	I/O
OCCI Timestamp	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 5-5 (Cont.) Data Conversions Between External and Internal Data Types

NA[1]	Internal Data Types									
External Data Types	**VARCHAR2**	**NUMBER**	**LONG**	**ROWID**	**DATE**	**RAW**	**LONG RAW**	**CHAR**	**BFLOAT**	**BDOUBLE**
RAW	I/O[6]	NA	I[5, 6]	NA	NA	I/O	I/O	I/O[6]	NA	NA
ROWID	I	NA	I	I/O	NA	NA	NA	I	NA	NA
STL string	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3]	-	I/O[4]	I/O[4]
STRING	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3, 5]	I/O	I/O	I/O
UNSIGNED	I/O[4]	I/O	I	NA	NA	NA	NA	I/O[4]	I/O	I/O
VARCHAR	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3]	NA	I/O	I/O
VARCHAR2	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3, 5]	I/O	I/O	I/O
VARNUM	I/O[4]	I/O	I	NA	NA	NA	NA	I/O[4]	I/O	I/O
VARRAW	I/O[6]	NA	I[5, 6]	NA	NA	I/O	I/O	I/O[6]	NA	NA

1 NA means not applicable.

2 I/O = Conversion is valid for input and output, unless otherwise specified.

This section includes the following topics:

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

5.4.1 Data Conversions for LOB Data Types

Table 5-6 Data Conversions for LOBs

EXTERNAL DATATYPES	INTERNAL DATATYPES	
	CLOB	**BLOB**
VARCHAR	I/O[1]	NA[2]
CHAR	I/O	NA
LONG	I/O	NA
LONG VARCHAR	I/O	NA
STL String	I/O	NA
RAW	NA	I/O
VARRAW	NA	I/O
LONG RAW	NA	I/O
LONG VARRAW	NA	I/O
OCCI Bytes	NA	I/O

1 I/O = Conversion is valid for input and output.

2 NA means not applicable.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle AI Database SecureFiles and Large Objects Developer's Guide* for an introduction to `LOB` data types.

5.4.2 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.

Table 5-7 Data Conversions for Date, Timestamp, and Interval Data Types

External Types	Internal Types						
NA[1]	VARCHAR, CHAR	DATE	TS	TSTZ	TSLTZ	INTERVAL YEAR TO MONTH	INTERVAL DAY TO SECOND
VARCHAR2, CHAR	I/O[2]	I/O	I/O	I/O	I/O	I/O	I/O
STL String	I/O	I/O	I/O	I/O	I/O	I/O	I/O
DATE	I/O	I/O	I/O	I/O	I/O	NA	NA
OCCI Date	I/O	I/O	I/O	I/O	I/O	NA	NA
ANSI DATE	I/O	I/O	I/O	I/O	I/O	NA	NA
TIMESTAMP (TS)	I/O	I/O	I/O	I/O	I/O	NA	NA
OCCI Timestamp	I/O	I/O	I/O	I/O	I/O	NA	NA
TIMESTAMP WITH TIME ZONE (TSTZ)	I/O	I/O	I/O	I/O	I/O	NA	NA
TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)	I/O	I/O	I/O	I/O	I/O	NA	NA
INTERVAL YEAR TO MONTH	I/O	NA	NA	NA	NA	I/O	NA
OCCI IntervalYM	I/O	NA	NA	NA	NA	I/O	NA
INTERVAL DAY TO SECOND	I/O	NA	NA	NA	NA	NA	I/O
OCCI IntervalDS	I/O	NA	NA	NA	NA	NA	I/O

1 NA means not applicable.

2 I/O = Conversion is valid for input and output.

These consideration 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- (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, 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

6
Metadata

This chapter describes how to retrieve metadata about result sets or the database as a whole.

This chapter contains these topics:

- Overview of Metadata
- Using Identity Column Metadata
- About Describing Database Metadata
- Attribute Reference Information

6.1 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 `get`*`xxx`*`()` methods to access this information.

> **See Also**
>
> Table 13-27

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 `OCCI_TYPECODE` constants. Internal PL/SQL types (boolean, indexed table) are not supported

Copyright © 1999, 2025, Oracle and/or its affiliates.

- The ATTR_DATA_TYPE returns types that represent the data types of the database columns. These values are of enumerated type Type. For example, LONG types return OCCI_SQLT_LNG types.

6.2 Using Identity Column Metadata

Starting with Oracle Database Release 12*c*, columns may be created as identity columns. When new rows are inserted into tables, the values for these columns are generated automatically.

This feature adds a new ColumnAttrId enum to the MetaData Class (see Table 13-27), and an overloaded form of getBoolean() method in the MetaData Class. Example 6-1 shows how to use this feature.

For more information, see *Oracle AI Database SQL Translation and Migration Guide*, and *Oracle AI Database SQL Language Reference*. Additionally, see the changes to *Oracle AI Database Reference*:

- A new IDENTITY_COLUMN column for views ALL_TAB_COLUMNS, DBA_TAB_COLUMNS, USER_TAB_COLUMNS, ALL_TAB_COLS, DBA_TAB_COLS, and USER_TAB_COLS
- A new HAS_IDENTITY column for views ALL_TABLES, DBA_TABLES, and USER_TABLES
- The new views ALL_TAB_IDENTITY_COLS, DBA_TAB_IDENTITY_COLS, and USER_TAB_IDENTITY_COLS, which display a table's identity column properties

Example 6-1 How to use Identity Column Metadata

```
vector<MetaData> v1;
MetaData metaData = conn->getMetaData(tableName);
columnCount = metaData.getInt(MetaData::ATTR_NUM_COLS);
cout << "Number of Columns : " << columnCount << endl;

v1 =  metaData.getVector(MetaData::ATTR_LIST_COLUMNS);

for(int i=0; i < v1.size(); i++) {
   MetaData md = v1[i];
   colNames[i] = md.getString(MetaData::ATTR_NAME);
   size[i] =  md.getInt(MetaData::ATTR_DATA_SIZE);
   precision[i] = md.getInt(MetaData::ATTR_PRECISION);
   scale[i] =  md.getInt(MetaData::ATTR_SCALE);

   if ( md.getBoolean(MetaData::ATTR_IS_NULL) )
      strcpy (isnull[i], "YES");
   else
      strcpy (isnull[i], "NO");

   if (md.getBoolean(MetaData::ATTR_COL_IS_IDENTITY))
      strcpy (isIdentity[i], "YES");
   else
      strcpy (isIdentity[i], "NO");

   if ( md.getBoolean(MetaData::ATTR_COL_IS_GEN_ALWAYS))
      strcpy (isGenAlways[i], "YES");
   else
      strcpy (isGenAlways[i], "NO");

   if (md.getBoolean(MetaData::ATTR_COL_IS_GEN_BY_DEF_ON_NULL))
      strcpy (isGenOnNull[i], "YES");
   else
      strcpy (isGenOnNull[i], "NO");
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
}

cout << "\n columnName  isNull   isIdentity   isGenAlways"  << "  isGenOnNull "
   << endl;
cout <<"-------------------------------------------------------" << endl;

for(int i=0; i < columnCount; ++i) {
   cout << "   " << colNames[i] << "  ";
   printf("%10s%10s%12s%12s\n", isnull[i], isIdentity[i], isGenAlways[i],
      isGenOnNull[i]);
}
```

6.3 About 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 `get`*`xxx`*`()` 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 `get`*`xxx`*`()` method.

Table 6-1 Attribute Groupings

Attribute Type	Description
Parameter Attributes	Attributes belonging to all elements
Table and View Attributes	Attributes belonging to tables and views
Procedure, Function, and Subprogram Attributes	Attributes belonging to procedures, functions, and package subprograms
Package Attributes	Attributes belonging to packages
Type Attributes	Attributes belonging to types
Type Attribute Attributes	Attributes belonging to type attributes
Type Method Attributes	Attributes belonging to type methods

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 6-1 (Cont.) Attribute Groupings

Attribute Type	Description
Collection Attributes	Attributes belonging to collection types
Synonym Attributes	Attributes belonging to synonyms
Sequence Attributes	Attributes belonging to sequences
Column Attributes	Attributes belonging to columns of tables or views
Argument and Result Attributes	Attributes belonging to arguments / results
List Attributes	Attributes that designate the list type
Schema Attributes	Attributes specific to schemas
Database Attributes	Attributes specific to databases

This section includes the following topic: Using Metadata (Code Examples).

6.3.1 Using Metadata (Code Examples)

This section provides code examples for using metadata:

- Example 6-2
- Example 6-3
- Example 6-4
- Example 6-5
- Example 6-6
- Example 6-7

Example 6-2 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:

```
/* Create an environment and a connection to the HR database */
.
.
/* Call the getMetaData method on the Connection object obtainedv*/
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)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
   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);

/* Each of the list elements represents a column metadata,
   so now you can access the column attributes*/
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 6-3 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.

```
/* 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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 6-4 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      about the object */
}
```

Example 6-5 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<MetaData>cmd=rset->getColumnListMetaData();

/* 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.

Example 6-6 How to Obtain Domain Information by `getMetaData()` on Table Name

```
//Domain Information by getMetaData() on table name
MetaData tableMD;                 // MetaData of table
vector<MetaData> colListMD;       // MetaData of column list of table
MetaData colMD;                  // MetaData of column
tableMD = conn->getMetaData("ExampleTable");
colListMD =  tableMD.getVector(MetaData::ATTR_LIST_COLUMNS);
// MetaData of the columns
for(int i = 0; i < colListMD.size(); i++)
  {
   colMD = colListMD[i];
   ...
   cout << "Domain Name: " <<
            colMD.getString(MetaData::ATTR_DOMAIN_NAME) << endl;
   cout << "Domain Schema: " <<
            colMD.getString(MetaData::ATTR_DOMAIN_SCHEMA) << endl;
            ...
   }
```

Example 6-7 How to Obtain Domain Information by `getColumnListMetaData()` on `resultSet` Object.

```
//Domain Information by getColumnListMetaData() on resultSet object.
vector<MetaData> colListMD;     // MetaData of column list of table
MetaData colMD;                 // MetaData of column
string sel_string = "select * from EMP";
...
ResultSet *resultSet = stmt->executeQuery(sel_string);
colListMD = resultSet->getColumnListMetaData();
// MetaData of the columns
   for(int i = 0; i < colListMD.size(); i++)
   {
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
colMD = colListMD[i];
...
cout << "Domain Name: " <<
        colMD.getString(MetaData::ATTR_DOMAIN_NAME) << endl;
cout << "Domain Schema: " <<
colMD.getString(MetaData::ATTR_DOMAIN_SCHEMA) << endl;
...
}
```

6.4 Attribute Reference Information

This section describes the following attributes that belong to schema and subschema objects:

- 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

6.4.1 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 6-2 Attributes that Belong to All Elements

Attribute	Description	Attribute Data Type
ATTR_OBJ_ID	Object or schema ID	unsigned int
ATTR_OBJ_NAME	Object, schema, or database name	string
ATTR_OBJ_SCHEMA	Schema where object is located	string

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 6-2 (Cont.) Attributes that Belong to All Elements

Attribute	Description	Attribute Data Type
ATTR_OBJ_PTYPE	Type of information described by the parameter. Possible values are: PTYPE_TABLE, Table PTYPE_VIEW, View PTYPE_PROC, Procedure PTYPE_FUNC, Function PTYPE_PKG, Package PTYPE_TYPE, Type PTYPE_TYPE_ATTR, Attribute of a type PTYPE_TYPE_COLL, Collection type information PTYPE_TYPE_METHOD, A method of a type PTYPE_SYN, Synonym PTYPE_SEQ, Sequence PTYPE_COL, Column of a table or view PTYPE_ARG, Argument of a function or procedure PTYPE_TYPE_ARG, Argument of a type method PTYPE_TYPE_RESULT, Results of a method PTYPE_SCHEMA, Schema PTYPE_DATABASE, Database	int
ATTR_TIMESTAMP	The TIMESTAMP of the object this description is based on (Oracle DATE format).	Timestamp

The sections that follow list attributes specific to different types of elements.

6.4.2 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:

Table 6-3 Attributes that Belong to Tables or Views

Attribute	Description	Attribute Data Type
ATTR_OBJID	Object ID	unsigned int
ATTR_NUM_COLS	Number of columns	int
ATTR_LIST_COLUMNS	Column list (type PTYPE_LIST)	vector<MetaData>
ATTR_REF_TDO	REF to the object type that is being described	RefAny
ATTR_IS_TEMPORARY	Identifies whether the table or view is temporary	bool
ATTR_IS_TYPED	Identifies whether the table or view is typed	bool
ATTR_DURATION	Duration of a temporary table. Values can be: • DURATION_SESSION (session) • DURATION_TRANS (transaction) • DURATION_NULL (table not temporary)	int

Copyright © 1999, 2025, Oracle and/or its affiliates.

The additional attributes belonging to tables are described in Table 6-4.

Table 6-4 Attributes Specific to Tables

Attribute	Description	Attribute Data Type
ATTR_DBA	Data block address of the segment header	unsigned int
ATTR_TABLESPACE	Tablespace the table resides on	int
ATTR_CLUSTERED	Identifies whether the table is clustered	bool
ATTR_PARTITIONED	Identifies whether the table is partitioned	bool
ATTR_INDEX_ONLY	Identifies whether the table is index only	bool

6.4.3 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

Attribute	Description	Attribute Data Type
ATTR_LIST_ARGUMENTS	Argument list; refer to List Attributes .	vector<MetaData>
ATTR_IS_INVOKER_RIGHTS	Identifies whether the procedure or function has invoker's rights.	int

The additional attributes belonging to package subprograms are described in Table 6-6.

Table 6-6 Attributes that Belong to Package Subprograms

Attribute	Description	Attribute Data Type
ATTR_NAME	Name of procedure or function	string
ATTR_OVERLOAD_ID	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.	int

6.4.4 Package Attributes

A parameter for a package (type PTYPE_PKG) has the type-specific attributes described in Table 6-7.

Table 6-7 Attributes that Belong to Packages

Attribute	Description	Attribute Data Type
ATTR_LIST_SUBPROGRAMS	Subprogram list; refer to List Attributes.	vector<MetaData>
ATTR_IS_INVOKER_RIGHTS	Identifies whether the package has invoker's rights	bool

Copyright © 1999, 2025, Oracle and/or its affiliates.

6.4.5 Type Attributes

A parameter for a type (type PTYPE_TYPE) has attributes described in Table 6-8.

Table 6-8 Attributes that Belong to Types

Attribute	Description	Attribute Data Type
ATTR_REF_TDO	Returns the in-memory ref of the type descriptor object for the type, if the column type is an object type.	RefAny
ATTR_TYPECODE	Type code. Can be: • OCCI_TYPECODE_OBJECT • OCCI_TYPECODE_NAMED COLLECTION	int
ATTR_COLLECTION_TYPECODE	Type code of collection if type is collection; invalid otherwise. Can be: • OCCI_TYPECODE_VARRAY • OCCI_TYPECODE_TABLE	int
ATTR_VERSION	A NULL-terminated string containing the user-assigned version	string
ATTR_IS_FINAL_TYPE	Identifies whether this is a final type	bool
ATTR_IS_INSTANTIABLE_TYPE	Identifies whether this is an instantiable type	bool
ATTR_IS_SUBTYPE	Identifies whether this is a subtype	bool
ATTR_SUPERTYPE_SCHEMA_NAME	Name of the schema containing the supertype	string
ATTR_SUPERTYPE_NAME	Name of the supertype	string
ATTR_IS_INVOKER_RIGHTS	Identifies whether this type is invoker's rights	bool
ATTR_IS_INCOMPLETE_TYPE	Identifies whether this type is incomplete	bool
ATTR_IS_SYSTEM_TYPE	Identifies whether this is a system type	bool
ATTR_IS_PREDEFINED_TYPE	Identifies whether this is a predefined type	bool
ATTR_IS_TRANSIENT_TYPE	Identifies whether this is a transient type	bool
ATTR_IS_SYSTEM_GENERATED_TYPE	Identifies whether this is a system-generated type	bool
ATTR_HAS_NESTED_TABLE	Identifies whether this type contains a nested table attribute	bool

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 6-8 (Cont.) Attributes that Belong to Types

Attribute	Description	Attribute Data Type
ATTR_HAS_LOB	Identifies whether this type contains a LOB attribute	bool
ATTR_HAS_FILE	Identifies whether this type contains a FILE attribute	bool
ATTR_COLLECTION_ELEMENT	Handle to collection element Refer to Collection Attributes	MetaData
ATTR_NUM_TYPE_ATTRS	Number of type attributes	unsigned int
ATTR_LIST_TYPE_ATTRS	List of type attributes Refer to List Attributes	vector<MetaData>
ATTR_NUM_TYPE_METHODS	Number of type methods	unsigned int
ATTR_LIST_TYPE_METHODS	List of type methods Refer to List Attributes	vector<MetaData>
ATTR_MAP_METHOD	Map method of type Refer to Type Method Attributes	MetaData
ATTR_ORDER_METHOD	Order method of type; refer to Type Method Attributes	MetaData

6.4.6 Type Attribute Attributes

A parameter for an attribute of a type (type PTYPE_TYPE_ATTR) has the attributes described in Table 6-9.

Table 6-9 Attributes that Belong to Type Attributes

Attribute	Description	Attribute Data Type
ATTR_DATA_SIZE	Maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.	int
ATTR_TYPECODE	Type code	int
ATTR_DATA_TYPE	Data type of the type attribute	int
ATTR_NAME	A pointer to a string that is the type attribute name	string
ATTR_PRECISION	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.	int
ATTR_SCALE	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.	int

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 6-9 (Cont.) Attributes that Belong to Type Attributes

Attribute	Description	Attribute Data Type
ATTR_TYPE_NAME	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.	string
ATTR_SCHEMA_NAME	String with the schema name under which the type has been created	string
ATTR_REF_TDO	Returns the in-memory REF of the TDO for the type, if the column type is an object type.	RefAny
ATTR_CHARSET_ID	Character set ID, if the type attribute is of a string or character type	int
ATTR_CHARSET_FORM	Character set form, if the type attribute is of a string or character type	int
ATTR_FSPRECISION	The fractional seconds precision of a datetime or interval	int
ATTR_LFPRECISION	The leading field precision of an interval	int

6.4.7 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

Attribute	Description	Attribute Data Type
ATTR_NAME	Name of method (procedure or function)	string
ATTR_ENCAPSULATION	Encapsulation level of the method; can be: • OCCI_TYPEENCAP_PRIVATE • OCCI_TYPEENCAP_PUBLIC)	int
ATTR_LIST_ARGUMENTS	Argument list	vector<MetaData>
ATTR_IS_CONSTRUCTOR	Identifies whether the method is a constructor	bool
ATTR_IS_DESTRUCTOR	Identifies whether the method is a destructor	bool
ATTR_IS_OPERATOR	Identifies whether the method is an operator	bool
ATTR_IS_SELFISH	Identifies whether the method is selfish	bool
ATTR_IS_MAP	Identifies whether the method is a map method	bool
ATTR_IS_ORDER	Identifies whether the method is an order method	bool
ATTR_IS_RNDS	Identifies whether "Read No Data State" is set for the method	bool

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 6-10 (Cont.) Attributes that Belong to Type Methods

Attribute	Description	Attribute Data Type
ATTR_IS_RNPS	Identifies whether "Read No Process State" is set for the method	bool
ATTR_IS_WNDS	Identifies whether "Write No Data State" is set for the method	bool
ATTR_IS_WNPS	Identifies whether "Write No Process State" is set for the method	bool
ATTR_IS_FINAL_METHOD	Identifies whether this is a final method	bool
ATTR_IS_INSTANTIABLE_METHOD	Identifies whether this is an instantiable method	bool
ATTR_IS_OVERRIDING_METHOD	Identifies whether this is an overriding method	bool

6.4.8 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

Attribute	Description	Attribute Data Type
ATTR_DATA_SIZE	Maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.	int
ATTR_TYPECODE	Typecode.	int
ATTR_DATA_TYPE	The data type of the type attribute.	int
ATTR_NUM_ELEMENTS	Number of elements in an array; only valid for collections that are arrays.	unsigned int
ATTR_NAME	A pointer to a string that is the type attribute name	string
ATTR_PRECISION	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 as NUMBER.	int
ATTR_SCALE	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.	int
ATTR_TYPE_NAME	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	string
ATTR_SCHEMA_NAME	String with the schema name under which the type has been created	string

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 6-11 (Cont.) Attributes that Belong to Collection Types

Attribute	Description	Attribute Data Type
ATTR_REF_TDO	Returns the in memory REF of the TDO for the type.	RefAny
ATTR_CHARSET_ID	Typecode.	int
ATTR_CHARSET_FORM	The data type of the type attribute.	int

6.4.9 Synonym Attributes

A parameter for a synonym (type PTYPE_SYN) has the attributes described in Table 6-12.

Table 6-12 Attributes that Belong to Synonyms

Attribute	Description	Attribute Data Type
ATTR_OBJID	Object ID	unsigned int
ATTR_SCHEMA_NAME	Null-terminated string containing the schema name of the synonym translation	string
ATTR_NAME	Null-terminated string containing the object name of the synonym translation	string
ATTR_LINK	Null-terminated string containing the database link name of the synonym translation	string

6.4.10 Sequence Attributes

A parameter for a sequence (type PTYPE_SEQ) has the attributes described in Table 6-13.

Table 6-13 Attributes that Belong to Sequences

Attribute	Description	Attribute Data Type
ATTR_OBJID	Object ID	unsigned int
ATTR_MIN	Minimum value (in Oracle number format)	Number
ATTR_MAX	Maximum value (in Oracle number format)	Number
ATTR_INCR	Increment (in Oracle number format)	Number
ATTR_CACHE	Number of sequence numbers cached; zero if the sequence is not a cached sequence (in Oracle number format)	Number
ATTR_ORDER	Identifies whether the sequence is ordered	bool
ATTR_HW_MARK	High-water mark (in Oracle number format)	Number

6.4.11 Column Attributes

A parameter for a column of a table or view (type PTYPE_COL) has the attributes described in Table 6-14.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 6-14 Attributes that Belong to Columns of Tables or Views

Attribute	Description	Attribute Data Type
ATTR_DATA_SIZE	Maximum size of the column. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER..	int
ATTR_DATA_TYPE	The data type of the column.	Type
ATTR_NAME	Pointer to a string that is the column name.	string
ATTR_PRECISION	Returns the precision.	int
ATTR_SCALE	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.	int
ATTR_IS_NULL	Returns FALSE if null values are not permitted for the column.	bool
ATTR_TYPE_NAME	Returns a string that is the type name. The returned value contains the type name if the data type is OCCI_SQLT_NTY or OCCI_SQLT_REF. If the data type is OCCI_SQLT_NTY, then the name of the named data type's type is returned. If the data type is OCCI_SQLT_REF, then the type name of the named data type pointed to by the REF is returned.	string
ATTR_SCHEMA_NAME	Returns a string with the schema name under which the type has been created.	string
ATTR_REF_TDO	The REF of the TDO for the type, if the column type is an object type.	RefAny
ATTR_CHARSET_ID	Character set ID for character column. If not set, the character set ID defaults to the character set ID set in the direct path context.	int
ATTR_CHARSET_FORM	Character set form of the column. Setting this attribute specifies the use of the database or national character set on the client side.	int

6.4.12 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

Attribute	Description	Attribute Data Type
ATTR_NAME	Returns a pointer to a string which is the argument name	string
ATTR_POSITION	Position of the argument in the argument list. Always returns 0.	int
ATTR_TYPECODE	Typecode.	int
ATTR_DATA_TYPE	Data type of the argument.	int

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 6-15 (Cont.) Attributes that Belong to Arguments / Results

Attribute	Description	Attribute Data Type
ATTR_DATA_SIZE	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.	int
ATTR_PRECISION	Precision of numeric arguments. 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.	int
ATTR_SCALE	Scale of numeric arguments. 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.	int
ATTR_LEVEL	Data type levels. This attribute always returns 0.	int
ATTR_HAS_DEFAULT	Indicates whether an argument has a default	int
ATTR_LIST_ARGUMENTS	The list of arguments at the next level (when the argument is of a record or table type)	vector<MetaData>
ATTR_IOMODE	Indicates the argument mode; valid values are: • 0 for IN (OCCI_TYPEPARAM_IN) • 1 for OUT (OCCI_TYPEPARAM_OUT) • 2 for IN/OUT (OCCI_TYPEPARAM_INOUT)	int
ATTR_RADIX	Returns a radix (if number type)	int
ATTR_IS_NULL	Returns FALSE if NULL values are not permitted for the column.	bool
ATTR_TYPE_NAME	Returns a string that is the type name (or the package name for local package types). 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.	string
ATTR_SCHEMA_NAME	For SQLT_NTY or SQLT_REF, returns a string with the schema name under which the type was created (or for local package types, the package name).	string
ATTR_SUB_NAME	For SQLT_NTY or SQLT_REF, returns a string with the type name.	string
ATTR_LINK	For SQLT_NTY or SQLT_REF, returns a string with the database link name of the database on which the type exists. This can happen only for package-ocal types, when the package is remote.	string
ATTR_REF_TDO	Returns the REF of the TDO for the type, if the argument type is an object.	RefAny
ATTR_CHARSET_ID	Returns the character set ID if the argument is of a string or character type.	int
ATTR_CHARSET_FORM	Returns the character set form if the argument is of a string or character type.	int

Copyright © 1999, 2025, Oracle and/or its affiliates.

6.4.13 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

Possible Values	Description
`ATTR_LIST_COLUMNS`	Column list
`ATTR_LIST_ARGUMENTS`	Procedure or function arguments list
`ATTR_LIST_SUBPROGRAMS`	Subprogram list
`ATTR_LIST_TYPE_ATTRIBUTES`	Type attribute list
`ATTR_LIST_TYPE_METHODS`	Type method list
`ATTR_LIST_OBJECTS`	Object list within a schema
`ATTR_LIST_SCHEMAS`	Schema list within a database

6.4.14 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

Attribute	Description	Attribute Data Type
`ATTR_LIST_OBJECTS`	List of objects in the schema	`string`

6.4.15 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

Attribute	Description	Attribute Data Type
`ATTR_VERSION`	Database version	`string`
`ATTR_CHARSET_ID`	Database character set ID from the server handle	`int`
`ATTR_NCHARSET_ID`	Database native character set ID from the server handle	`int`
`ATTR_LIST_SCHEMAS`	List of schemas (type `PTYPE_SCHEMA`) in the database	`vector<MetaData>`
`ATTR_MAX_PROC_LEN`	Maximum length of a procedure name	`unsigned int`
`ATTR_MAX_COLUMN_LEN`	Maximum length of a column name	`unsigned int`

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 6-18 (Cont.) Attributes Specific to Databases

Attribute	Description	Attribute Data Type
ATTR_CURSOR_COMMIT_BEHAVIOR	How a COMMIT operation affects cursors and prepared statements in the database; values are: • OCCI_CURSOR_OPEN for preserving cursor state as before the commit operation • 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	int
ATTR_MAX_CATALOG_NAMELEN	Maximum length of a catalog (database) name	int
ATTR_CATALOG_LOCATION	Position of the catalog in a qualified table; values are: • OCCI_CL_START • OCCI_CL_END	int
ATTR_SAVEPOINT_SUPPORT	Identifies whether the database supports savepoints; values are: • OCCI_SP_SUPPORTED • OCCI_SP_UNSUPPORTED	int
ATTR_NOWAIT_SUPPORT	Identifies whether the database supports the nowait clause; values are: • OCCI_NW_SUPPORTED • OCCI_NW_UNSUPPORTED	int
ATTR_AUTOCOMMIT_DDL	Identifies whether the autocommit mode is required for DDL statements; values are: • OCCI_AC_DDL • OCCI_NO_AC_DDL	int
ATTR_LOCKING_MODE	Locking mode for the database; values are: • OCCI_LOCK_IMMEDIATE • OCCI_LOCK_DELAYED	int

Copyright © 1999, 2025, Oracle and/or its affiliates.

7

Programming with LOBs

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
- About Reading and Writing LOBs
- Using Objects with LOB Attributes
- About Using SecureFiles

> **See also**
>
> *Oracle AI Database SecureFiles and Large Objects Developer's Guide* for extensive information about LOBs

7.1 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.

This section includes the following topics:

- Introducing Internal LOBs
- Introducing External LOBs
- About Storing LOBs

7.1.1 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 AI database
- `NCLOB`: A LOB whose value is composed of character data that corresponds to the national character set defined for the Oracle AI Database

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

7.1.2 Introducing External LOBs

`BFILE`s 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, `BFILE`s 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 `BFILE`s 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.

7.1.3 About 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.

7.2 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 `BFILE`s, 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 OCCI Application Programming Interface:
  - Bfile Class contains the APIs for `BFILE`s, as summarized in Table 13-7.
  - Blob Class contains the APIs for `BLOB`s, as summarized in Table 13-8.

Copyright © 1999, 2025, Oracle and/or its affiliates.

  - Clob Class 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 extensive 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.

7.3 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.

7.4 About Reading and Writing LOBs

There are two general methods for reading and writing LOBs: non-streamed, and streamed.

This section includes the following topics:

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

7.4.1 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 7-2 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.

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 7-1 How to Read Non-Streamed BLOBs

```
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 How to Read Non-Streamed BFILESs

```
ResultSet *rset=stmt->executeQuery("SELECT ad_graphic FROM print_media
                                    WHERE product_id=6666");
while(rset->next())
{
   Bfile file=rset->getBfile(1);
   if(bfile.isNull())
      cerr <<"Null Bfile"<<endl;
   else
   {
      //display the directory alias and the file name of the BFILE
      cout <<"File Name:"<<bfile.getFileName()<<endl;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      cout <<"Directory Alias:"<<bfile.getDirAlias()<<endl;

      if(bfile.fileExists())
      {
         unsigned int length=bfile.length();
         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 7-3 How to Read Streamed BLOBs

```
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
   {
      Stream *instream=blob.getStream(1,0);
      //reading from offset 1 to the end of the BLOB

      unsigned int size=blob.getChunkSize();
      char *buffer=new char[size];

      while((unsigned int length=instream->readBuffer(buffer,size))!=-1)
      {
         //process "length" bytes read into buffer
         ...
      }
      delete[] buffer;
      blob.closeStream(instream);
   }
}
stmt->closeResultSet(rset);
```

7.4.2 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 7-5 demonstrates how to write data to an internal LOB that is populated by using a streamed write.

Example 7-4 How to Write Non-Streamed BLOBs

```
ResultSet *rset=stmt->executeQuery("SELECT ad_composite FROM print_media
                                    WHERE product_id=6666 FOR UPDATE");
while(rset->next())
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
{
   Blob blob=rset->getBlob(1);
   if(blob.isNull())
      cerr <<"Null Blob"<<endl;
   else
   {
      blob.open(OCCI_LOB_READWRITE);

      const unsigned int BUFSIZE=100;
      char buffer[BUFSIZE];
      unsigned int writeAmt=BUFSIZE;
      unsigned int offset=1;

      //writing writeAmt bytes from offset 1
      //contents of buffer are replaced after each writeChunk(),
      //typically with an fread()
      while(<fread "BUFSIZE" bytes into buffer succeeds>)
      {
         blob.writeChunk(writeAmt, buffer, BUFSIZE, offset);
         offset += writeAmt;
      }
      blob.writeChunk(<remaining amt>, buffer, BUFSIZE, offset);

      blob.close();
   }
}
stmt->closeResultSet(rset);
conn->commit();
```

Example 7-5 How to Write Streamed BLOBs

```
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>)
         ostream->writeBuffer(buffer,BUFSIZE);
      ostream->writeLastBuffer(buffer,<remaining amt>);
      blob.closeStream(outstream);
   }
}
stmt->closeResultSet(rset);
conn->commit();
```

7.4.3 About Enhancing the Performance of LOB Reads and Writes

Reading and writing of internal LOBs can be improved by using either `getChunkSize()` method.

This section includes the following topic: About Using the getChunkSize() Method.

Copyright © 1999, 2025, Oracle and/or its affiliates.

7.4.3.1 About 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.

7.4.4 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();
```

7.4.5 About Reading and Writing Multiple LOBs

As of Oracle Database 10*g* 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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 OCCI Application Programming Interface, section on Connection Class, and include readVectorOfBfiles(), readVectorOfBlobs(), readVectorOfClobs() (overloaded to support general charactersets, and the `UTF16` characterset in particular), writeVectorOfBlobs(), and writeVectorOfClobs() (overloaded to support general charactersets, and the `UTF16` characterset in particular).

This section includes the following topic: About Using the Interfaces for Reading and Writing Multiple LOBs.

7.4.5.1 About Using the Interfaces for Reading and Writing Multiple LOBs

Each of the `readVectorOf`*`xxx`*`()` and `writeVectorOf`*`xxx`*`()` 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 `Clobs` 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.

7.5 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();
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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( ... );
```

7.6 About Using SecureFiles

Introduced with Oracle Database 11*g* Release 1, SecureFiles LOBs add powerful new features for LOB compression, encryption, and deduplication.

> **See Also**
>
> *Oracle AI Database SecureFiles and Large Objects Developer's Guide*

This section includes the following topics:

- About Using SecureFile Compression
- About Using SecureFiles Encryption
- About Using SecureFiles Deduplication
- About Combining SecureFiles Compression, Encryption, and Deduplication
- SecureFiles LOB Types and Constants

Copyright © 1999, 2025, Oracle and/or its affiliates.

7.6.1 About 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.

7.6.2 About 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.

7.6.3 About Using SecureFiles Deduplication

SecureFiles deduplication enables Oracle AI Database to automatically detect duplicate LOB data, and to conserve space by storing a single copy of the SecureFiles LOB.

7.6.4 About Combining SecureFiles Compression, Encryption, and Deduplication

You can combine compression, encryption and deduplication in any combination. Oracle AI 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.

7.6.5 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 7-1 Values of Type LobOptionType

Value	Description
`OCCI_LOB_OPT_COMPRESS`	Compression option type
`OCCI_LOB_OPT_ENCRYPT`	Encryption option type
`OCCI_LOB_OPT_DEDUPLICATE`	Deduplicate option type

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 7-2 Values of Type LobOptionValue

Value	Description
`OCCI_LOB_COMPRESS_OFF`	Turns off SecureFiles compression
`OCCI_LOB_COMPRESS_ON`	Turns on SecureFiles compression
`OCCI_LOB_ENCRYPT_OFF`	Turns off SecureFiles encryption
`OCCI_LOB_ENCRYPT_ON`	Turns on SecureFiles encryption
`OCCI_LOB_DEDUPLICATE_OFF`	Turns off SecureFiles deduplication
`OCCI_LOB_DEDUPLICATE_ON`	Turns off LOB deduplication

Copyright © 1999, 2025, Oracle and/or its affiliates.

8

Object Type Translator Utility

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
- About Invoking the OTT Utility
- About Using the INTYPE File
- Using OTT Utility Data Type Mappings
- Overview of the OUTTYPE File
- Using 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.

8.1 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 AI 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
- Include the header file containing the prototype for the function to register the mappings

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Link with the C++ source file containing the static methods to be called by OCCI while instantiating the objects
- Link with the file containing the function to register the mappings with the environment and call this function

8.2 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.

> **See Also**
>
> Extending C++ Classes for a complete C++ example

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 user name `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
```

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`.

8.3 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));
```

8.4 About Invoking the OTT Utility

After creating types in the database, the next step is to invoke the OTT utility.

This section includes the following topics:

- Specifying OTT Parameters

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Invoking the OTT Utility on the Command Line
- OTT Utility Parameters
- Where OTT Parameters Can Appear
- File Name Comparison Restriction
- Using the OTT Command on Microsoft Windows

8.4.1 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.

This section includes the following topics:

- About Setting Parameters on the Command Line
- About Setting Parameters in the INTYPE File
- About Setting Parameters in the Configuration File

8.4.1.1 About Setting Parameters on the Command Line

Parameters (also called options) set on the command line override any parameters or option set elsewhere.

8.4.1.2 About 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.4.1.3 About 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.

8.4.2 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.

> **See Also**
>
> Your operating system-specific documentation to see how to invoke the OTT utility on your operating system

> **Note**
>
> 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.

Example 8-3 How to Invoke the OTT Utility to Generate C++ Classes

OTT attempts to connect with user name `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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

This section includes the following topic: Elements Used on the OTT Command Line.

8.4.2.1 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
- INTYPE parameter
- OUTTYPE parameter.
- CODE parameter.
- HFILE parameter.
- CPPFILE parameter.
- MAPFILE parameter.

8.4.3 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 8-1 Summary of OTT Utility Parameters

Parameter	Description
ATTRACCESS	Specifies whether the access to type attributes is `PROTECTED` or `PRIVATE`.
CASE	Affects the letter case of generated C++ identifiers
CODE	Specifies the target language for the translation. Use `CPP`.
CONFIG	Specifies the name of the OTT configuration file that lists commonly used parameter specifications.
CPPFILE	Specifies the name of the C++ source file into which the method implementations are written.
ERRTYPE	Specifies the name of the error message output file.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 8-1 (Cont.) Summary of OTT Utility Parameters

Parameter	Description
HFILE	Specifies the name of the C++ header file to which the generated C++ classes are written.
INTYPE	Specifies the name of the `INTYPE` file.
MAPFILE	Specifies the name of the mapping file and the corresponding header file generated by the OTT utility.
MAPFUNC	Specifies the name of the function used to register generated mappings.
OUTTYPE	Specifies the name of the `OUTTYPE` file.
SCHEMA_NAMES	Controls the qualifying the database name of a type from the default schema
TRANSITIVE	Indicates whether to translate type dependency that are not explicitly listed in the INTYPE.
UNICODE	Indicates whether the application should provide UTF16 support generate `UString` types.
USE_MARKER	Indicates whether OTT markers should be supported to carry forward user added cod
USERID	Specifies the database connection information that the OTT utility uses.

8.4.3.1 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, `get`*`XXX`*`()` and `set`*`XXX`*`()`.

8.4.3.2 CASE

This parameter affects the letter case of generated C++ identifiers. The valid values of `CASE` are:

- `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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.4.3.3 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.

8.4.3.4 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.

8.4.3.5 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 an 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.

8.4.3.6 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.

8.4.3.7 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.4.3.8 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" for more information about the format of the `INTYPE` file

8.4.3.9 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.

8.4.3.10 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.

8.4.3.11 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.4.3.12 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:

```
CREATE TYPE Address AS OBJECT
(
   street    VARCHAR2(40),
   city      VARCHAR(30),
   state     CHAR(2),
   zip_code  CHAR(10)
);

CREATE TYPE Person AS OBJECT
(
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
   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 `TYPE joe.Person` appears in the `INTYPE` file, then `TYPE joe.Person` and `TYPE joe.Address` appear 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.

8.4.3.13 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.

8.4.3.14 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 `get`*`XXX`*`()`/`set`*`XXX`*`()` 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

```
create type CitiesList as varray(100) of varchar2(100);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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

```
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

```
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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
...
}
```

8.4.3.15 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.

8.4.3.16 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.

8.4.4 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 `MAPFNC` 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

8.4.5 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 foo1.h, and another type declaration written to /private/smith/foo1.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:

```
#ifndef FOO1_ORACLE
#include "foo1.h"
#endif
#ifndef FOO1_ORACLE
#include "/private/smith/foo1.h"
#endif
```

If foo1.h and /private/smith/foo1.h are different files, then only the first one is included. If foo1.h and /private/smith/foo1.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.

8.4.6 Using the OTT Command on Microsoft Windows

OTT executable on Microsoft Windows in the current release is ott.bat, instead of ott.exe as in the earlier releases. This may break Windows batch scripts, as the scripts exit immediately after executing ott. To fix this problem, OTT should be invoked as follows, in Windows batch scripts:

```
call ott [arguments]
```

> **Note**
>
> ORACLE_HOME\precomp\admin\ott.exe can be used until the scripts are fixed, as an intermediate solution. However, this intermediate solution will not be provided in future releases.

8.5 About 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.

This section includes the following topics:

- Using the INTYPE File

Copyright © 1999, 2025, Oracle and/or its affiliates.

- Structure of the INTYPE File
- Using Nested include File Generation

8.5.1 Using the INTYPE File

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 to 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" for a more detailed specification of the structure of the `INTYPE` file and the available options.
> - "CASE" for further information regarding the `CASE` parameter

Example 8-8 How to Create a User Defined INTYPE File Using the OTT Utility

```
CASE=LOWER
TYPE employee
   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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

8.5.2 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 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" for an example of a simple user-defined INTYPE file and of the full OUTTYPE file that the OTT utility generates from it

This section includes the following topic: INTYPE File Type Specifications.

8.5.2.1 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- *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.
- *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.

8.5.3 Using 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:

```
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:

```
CASE=lower
type pxl
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
   hfile tott95a.h
type px3
   hfile tott95b.h
```

You invoke the OTT utility as follows:

```
>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

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`.

Example 8-9 Listing of ott95a.h

```
#ifndef TOTT95A_ORACLE
# define TOTT95A_ORACLE

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

/*************************************************************/
//  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,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      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

```
#ifndef TOTT95B_ORACLE
# define TOTT95B_ORACLE

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

#ifndef TOTT95A_ORACLE
# include "tott95a.h"
#endif

/*************************************************************/
//  generated declarations for the PX3 object type.
/*************************************************************/

class px3 : public oracle::occi::PObject {

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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

8.6 Using 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`).

Table 8-2 C++ Object Data Type Mappings for Object Type Attributes

Object Attribute Types	C++ Mapping
`BFILE`	`Bfile`
`BLOB`	`Blob`
`BINARY_DOUBLE`	`BDouble`
`BINARY_FLOAT`	`BFloat`
`CHAR(n), CHARACTER(n)`	`string`
`CLOB`	`Clob`
`DATE`	`Date`
`DEC, DEC(n), DEC(n,n)`	`Number`
`DECIMAL, DECIMAL(n), DECIMAL(n,n)`	`Number`
`FLOAT, FLOAT(n), DOUBLE PRECISION`	`Number`
`INT, INTEGER, SMALLINT`	`Number`
`INTERVAL DAY TO SECOND`	`IntervalDS`
`INTERVAL YEAR TO MONTH`	`IntervalYM`
`Nested Object Type`	C++ name of the nested object type
`NESTED TABLE`	`vector<attribute_type>`
`NUMBER, NUMBER(n), NUMBER(n,n)`	`Number`
`NUMERIC, NUMERIC(n), NUMERIC(n,n)`	`Number`
`RAW`	`Bytes`
`REAL`	`Number`
`REF`	`Ref<attribute_type>`
`TIMESTAMP,TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE`	`Timestamp`
`VARCHAR(n)`	`string`
`VARCHAR2(n)`	`string`
`VARRAY`	`vector<attribute_type>`

Copyright © 1999, 2025, Oracle and/or its affiliates.

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:

```
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:

```
#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, 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();

};

#endif
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

Example 8-12 How to Map Object Data Types Using the OTT Utility

The example assumes that the following database types are created:

```
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

#ifndef OCCI_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,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
         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,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
                              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_);
};

#endif
```

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.

This section includes the following topic: Default Name Mapping.

8.6.1 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

8.7 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.

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.

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:

```
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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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:

```
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`.

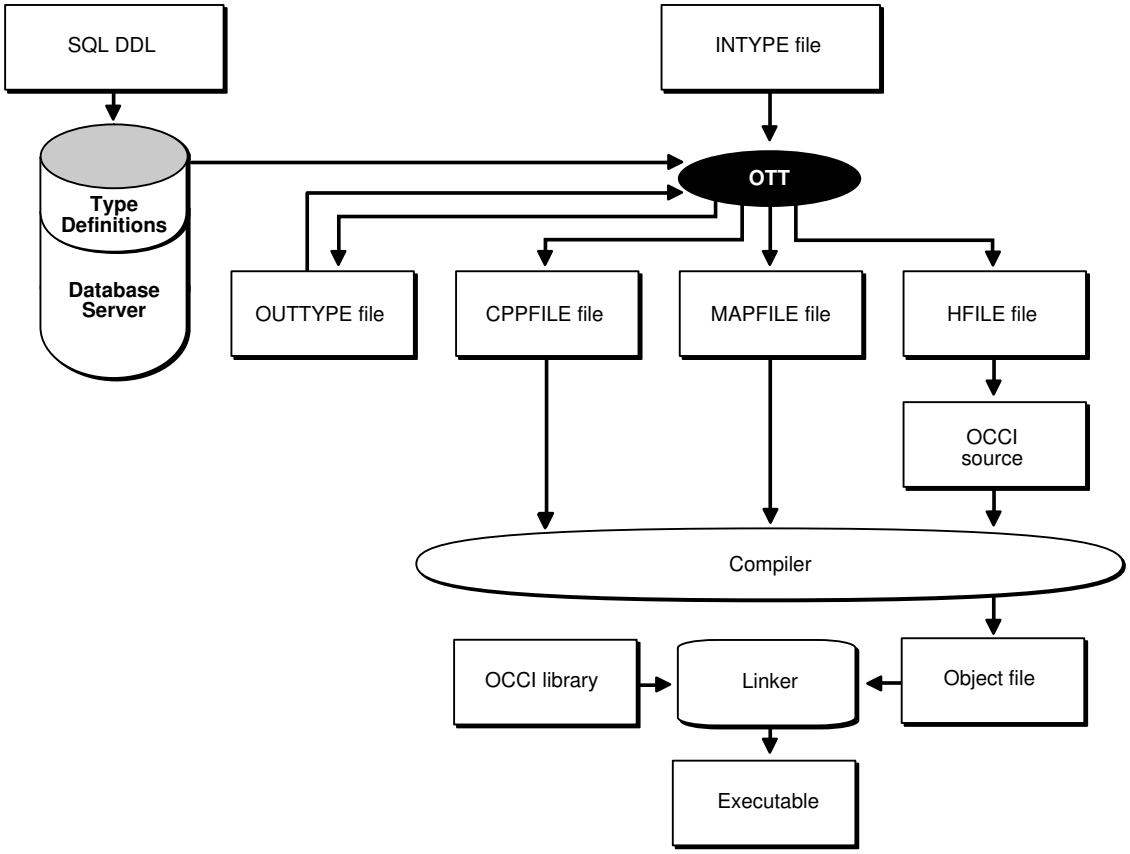
8.8 Using the OTT Utility and OCCI Applications

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

8.9 Generating 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.

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:

```
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_l 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
```

This section includes the following topics:

- Map Registry Function
- Extending C++ Classes
- Carrying Forward User Added Code

Copyright © 1999, 2025, Oracle and/or its affiliates.

8.9.1 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.

8.9.2 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" 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.

> **Note**
>
> When a class is both extended and used as a base class for another generated class, the *inheriting* type class and the *inherited* type class must be generated in separate files.

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 CAdress AS MyAddress
```

Given the database types `FULL_NAME`, `ADDRESS`, `PERSON`, and `PFGRFDENT` as they were created before and changing the `INTYPE` file to include the `GENERATE...AS` clause:

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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
```

8.9.3 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.

This section includes the following topics:

- How to Use Properties of OTT Markers
- Using OTT Markers

8.9.3.1 How to Use 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
   ```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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
TYPE employee
TRANSLATE SALARY$ AS salary
   DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

```
CASE=LOWER
TYPE employee
TRANSLATE SALARY$ AS salary
   DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

   - 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=LOWER
TYPE employee
TRANSLATE SALARY$ AS salary
   DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

```
CASE=LOWER
TYPE employee
TRANSLATE SALARY$ AS salary
   DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS
   purchase_order
```

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`

8.9.3.2 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.**

Copyright © 1999, 2025, Oracle and/or its affiliates.

- **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.

How to Add User Code to a Header File Using OTT Utility

```
...
#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 ...            // OTT generated include
#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);
      ...
      OTT_USERCODE_START   // user added code  for data member / method
      ...                  // declaration / inline method definition
      OTT_USERCODE_END
};

OTT_USERCODE_START     // user added code
...
OTT_USERCODE_END

class <class_name_2> : public oracle::occi::PObject
{
   ...
};

OTT_USERCODE_START     // user added code
...
OTT_USERCODE_END
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
...
#endif                  // end of .h file
```

2. **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.

   How to Add User Code to the Source File Using the OTT Utility

```
#ifndef OTT_USERCODE_START
#define OTT_USERCODE_START
#endif

#ifndef OTT_USERCODE_END
#define OTT_USERCODE_END
#endif
...
   OTT_USERCODE_START    // user added code
      ...
   OTT_USERCODE_END
...
   OTT_USERCODE_START    // user added code
      ...
   OTT_USERCODE_END

/*************************************************************
/ 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

9

Globalization and Unicode Support

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
- About Using Objects and OTT Support

9.1 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, user names, 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.

9.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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");
```

9.3 Data Types for Globalization and Unicode Support

The data types used for supporting globalization and use of unicode include:

- Using the UString Data Type
- Using Multibyte and UTF16 data
- Using CLOB and NCLOB Data Types

9.3.1 Using the UString Data Type

UString is a data type that enables applications and the OCCI 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.

OCCI applications should use the UString data type for data in UTF16 characterset

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"");
```

9.3.2 Using Multibyte and UTF16 data

For data in multibyte charactersets 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 characterset. Due to the use of string type, OCCI supports only byte length semantics for multibyte characterset strings.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 9-3 Binding UTF8 Data Using the string Data Type

```
//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 characterset, 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

```
//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
```

9.3.3 Using 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 characterset and NCLOBs represent data in the database national characterset. 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 charactersets. By default, the Clob interfaces assume the data is encoded in the client-side characterset (for both CLOBs and NCLOBs). To specify a different characterset or to specify the client-side national characterset for a NCLOB, call the setCharSetId() or setCharSetIdUString() methods with the appropriate characterset. The OCCI specific string 'OCCIUTF16' can be passed to indicate UTF16 as the characterset.

To read or write data in multibyte charactersets, 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.

Example 9-5 Using CLOB and NCLOB Data Types

```
//client characterset - ZHT16BIG5, national characterset - UTF16
Environment *env = Environment::createEnvironment("ZHT16BIG5","OCCIUTF16");...
Clob nclobvar;
//for NCLOBs, must call setCharSetForm method.
nclobvar.setCharSetForm(OCCI_SQLCS_NCHAR);...
//if reading/writing data in UTF16 for this NCLOB, still must
//explicitly call setCharSetId
nclobvar.setCharSetId("OCCIUTF16")
```

9.4 About Using Objects and OTT Support

Multibyte and UTF16 charactersets are supported for handling character data in object attributes. All CHAR data type (CHAR or VARCHAR2) attributes hold data in the client-side

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

- OCCI Application Programming Interface: two new versions of operator new() that have been added to the PObject Class for object support
- Object Type Translator Utility: a new UNICODE parameter that has been added for OTT utility support.

Copyright © 1999, 2025, Oracle and/or its affiliates.

10

Oracle Database Advanced Queuing

This chapter describes the OCCI implementation of Oracle Database Advanced Queuing (AQ) for messages.

This chapter contains these topics:

- Overview of Oracle Database Advanced Queuing
- About AQ Implementation in OCCI
- About Creating Messages
- Enqueuing Messages
- Dequeuing Messages
- Listening for Messages
- About Registering for Notification
- About Message Format Transformation

See Also

- *Oracle AI Database Transactional Event Queues and Advanced Queuing User's Guide* for basic concepts of Advanced Queuing
- OCCI Application Programming Interface

10.1 Overview of Oracle Database 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 Database Advanced Queuing (AQ).

Advanced Queuing is the integrated message queuing feature that exposes message queuing capabilities of Oracle Database AQ enables applications to:

- Perform message queuing operations similar to SQL operations from Oracle AI 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

10.2 About 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 AI 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 following abstractions:

- Message
- Agent
- Producer
- Consumer
- Listener
- Subscription

10.2.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Message Class documentation in OCCI Application Programming Interface

10.2.2 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.

> **See Also**
>
> Agent Class documentation in OCCI Application Programming Interface

Example 10-1 Creating an Agent

```
Agent agt(env, "Billing_app", "billqueue", 0);
```

10.2.3 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 OCCI Application Programming Interface

10.2.4 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,

> **See Also**
>
> Consumer Class documentation in OCCI Application Programming Interface

Example 10-2 Setting the Agent on the Consumer

```
Consumer cons(conn);
...
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
cons.setAgent(ag);
cons.receive();
```

10.2.5 Listener

A `Listener` listens for `Messages` for registered `Agents` at specified queues.

> **See Also**
>
> Listener Class documentation in OCCI Application Programming Interface

10.2.6 Subscription

A `Subscription` encapsulates the information and operations necessary for registering a subscriber for notifications.

10.3 About 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.

This section includes the following topics:

- About Message Payloads
- Message Properties

10.3.1 About Message Payloads

OCCI supports three types of message payloads:

- RAW
- AnyData
- Using User-defined Types as Payloads

10.3.1.1 RAW

`RAW` payloads are mapped as objects of the Bytes Class in OCCI.

10.3.1.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Example 10-3 Creating an AnyData Message with a String Payload

```
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

```
TypeCode tc = any.getType();
```

10.3.1.3 Using User-defined Types as Payloads

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

```
// 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);
```

10.3.2 Message Properties

Aside from payloads, the user can specify several additional message properties, such as:

- Correlation
- Sender
- Delay and Expiration
- Recipients
- Priority and Ordering

10.3.2.1 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");
```

10.3.2.2 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);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

10.3.2.3 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);
```

10.3.2.4 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);
```

10.3.2.5 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);
```

10.4 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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

10.5 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.

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 10-12 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 is assigned the content of the message
obj = mes.getObject();
```

Example 10-13 Receiving a Message

```
//receiving a RAW message
Message mes = cons.receive(Message::RAW);
...
//receiving an ANYDATA message
Message mes = cons.receive(Message::ANYDATA);
...
```

This section includes the following topic: About Dequeuing Options.

10.5.1 About Dequeuing Options

The dequeuing application can specify several dequeuing options before it begins to receive messages. These include:

- Correlation
- Mode
- Navigation

10.5.1.1 Correlation

The message can be dequeued based on the value of its correlation identifier using the `setCorrelationId()` method, as shown in Example 10-14.

Copyright © 1999, 2025, Oracle and/or its affiliates.

10.5.1.2 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.

10.5.1.3 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

```
cons.setCorrelationId(corrId);
...
cons.setDequeueMode(deqMode);
...
cons.setPositionOfMessage(Consumer::DEQ_NEXT_TRANSACTION);
```

10.6 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

```
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;
}
```

10.7 About 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

This section includes the following topics:

- Publish-Subscribe Notifications
- About Message Format Transformation

10.7.1 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:

- How to Use Direct Registration
- Using Open Registration

10.7.1.1 How to Use 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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::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()`.

10.7.1.2 Using 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):

  ```
  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
```

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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()`.

10.7.2 About 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 Database AQ namespace, this happens when a message of interest is enqueued.

The callback must return `0`, and it must have the following specification:

```
typedef unsigned int (*callbackfn) (Subscription &sub, NotifyResult *nr);
```

where:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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 10-1 Notification Result Attributes; ANONYMOUS and AQ Namespace

Notification Result Attribute	ANONYMOUS Namespace	AQ Namespace, Persistent Queue	AQ Namespace, Non-Persistent Queue
payload	valid	invalid	invalid
message	invalid	invalid	valid
messageID	invalid	valid	valid
consumer name	invalid	valid	valid
queue name	invalid	valid	valid

10.8 About 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 AI Database Transactional Event Queues and 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

11

Oracle XA Library

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:

- Developing Applications with XA and OCCI
- APIs for XA Support

See Also

- http://www.opengroup.org
- *Oracle Database Development Guide* for more details on the Oracle XA library and architecture
- OCCI Application Programming Interface

11.1 Developing Applications 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 AI 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, TMNOFLAGS);
/* 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->getXAConnection(
   "oracle_xa+acc=p/HR/password+sestm=10");

/* Use the Environment & Connection objects */
Statement *stmt = xaconn->createStatement(
   "Update Emp set sal = sal * 0.2");

...
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
/* Release the Environment & Connection objects */
xaenv->releaseXAConnection(xaconn);
Environment::releaseXAEnvironment(xaenv);
```

11.2 APIs for XA Support

The following methods of the Environment Class support use of XA libraries:

- getXAConnection()
- releaseXAEnvironment()
- releaseXAConnection()
- releaseXAEnvironment()

In addition, the getXAErrorCode() method of SQLException Class 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).

Copyright © 1999, 2025, Oracle and/or its affiliates.

12

Optimizing Performance of C++ Applications

This chapter describes a few suggestions that lead to better performance for your OCCI custom applications.

This chapter contains these topics:

- About Transparent Application Failover
- About Connection Sharing
- About Application-Managed Data Buffering
- Using the Array Fetch Using next() Method
- Modifying Rows Iteratively
- About Using Oracle Connection Manager in Traffic Director Mode
- About Run-time Load Balancing of the Stateless Connection Pool
- About Fault Diagnosability
- Using Client Result Cache
- About Client-Side Deployment Parameters and Auto Tuning

See Also

- OCCI Application Programming Interface

12.1 About 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 `ALTER SESSION` commands before the failover began, they are not automatically replayed on the second instance.

  Consequently, the developer may want to replay these `ALTER SESSION` commands on the second instance.

It is the user's responsibility to track changes to the `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 `ALTER SESSION` commands and inform the user that failover has happened. Note that you must keep track of `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 `FO_RETRY` to indicate that the failover should be attempted again.

> **See Also**
>
> – Definition of `FailOverType` and `FailOverEventType` in Table 13-11 in OCCI Application Programming Interface
>
> – *Oracle AI Database Net Services Administrator's Guide* for more detailed information about application failover.

This section includes the following topics:

- Using Transparent Application Failover
- About Objects and Transparent Application Failover
- Using Connection Pooling and Transparent Application Failover

12.1.1 Using Transparent Application Failover

To enable TAF, the connect string has to be configured for failover and registered on `Connection` (created from `Environment`, `ConnectionPool` and `StatelessConnectionPool`). To register the callback function, use the Connection Class interface setTAFNotify():

```
void Connection::setTAFNotify(
   int (*notifyFn)(
      Environment *env,
      Connection *conn,
      void *ctx,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    FailOverType foType,
    FailOverEventType foEvent),
void *ctxTAF);
```

Note that TAF support for `ConnectionPool`s does not include `BACKUP` and `PRECONNECT` clauses; these should not be used in the connect string.

12.1.2 About 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.

12.1.3 Using 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(), 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

```
#include <occi.h>
#include <iostream>
#include <time.h>

using namespace std;
using namespace oracle::occi;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
//Application custom pool of 3 connections
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 connnection " << 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");
 conn2 = env->createConnection("hr","password","inst1_failback");
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
 conn3 = env->createConnection("hr","password","inst1_failback");

//all connections are 'free'
conn1free = conn2free = conn3free = true;

//set TAF callbacks on all connection
conn1->setTAFNotify(taf_callback,NULL);
conn2->setTAFNotify(taf_callback,NULL);
conn3->setTAFNotify(taf_callback,NULL);

//use 1 connection
conn1free=false;
cout << "Using conn1" << endl;
Statement *stmt = conn1->createStatement ("select * from employees");
ResultSet *rs  = stmt->executeQuery();
while (rs->next())
{
   cout << (rs->getString(2)) << endl;
}
stmt->closeResultSet(rs);
conn1->terminateStatement(stmt);

cout << "Shutdown and restart the database" << endl;
string buf;
cin >> buf;

Statement *stmt2;
try
{
   cout << "Trying a update on EMP table" << endl;
   stmt2 = conn1->createStatement("delete from employees");
   stmt2->executeUpdate();
}
catch (SQLException &ex)
{
   cout << "Update EMPLOYEES returned error : " << ex.getMessage() << endl;
   cin >> buf;
}

cout << "Done" << endl;
env->terminateConnection(conn1);
env->terminateConnection(conn2);
env->terminateConnection(conn3);
Environment::terminateEnvironment(env);
}
catch(SQLException &ex)
{
cout << ex.getMessage() << endl;
}
}
```

12.2 About Connection Sharing

This section covers the following topics:

- Introduction to Thread Safety
- Implementing Thread Safety
- About Serialization
- Operating System Considerations

Copyright © 1999, 2025, Oracle and/or its affiliates.

12.2.1 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 Oracle AI 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 AI 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.

12.2.2 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 `THREADED_MUTEXED` or `THREADED_UNMUTEXED` for the mode parameter of the `createEnvironment()` method. For example, to turn on mutual exclusivity locking, issue the following statement:

```
Environment *env = Environment::createEnvironment(
          Environment::THREADED_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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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.

12.2.3 About 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

12.2.3.1 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.

12.2.3.2 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:

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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.

12.2.4 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" and "Implementing Thread Safety" for more information about threads.

For improving performance with many concurrently opened connections, see "About Pooling Connections".

12.3 About Application-Managed Data Buffering

When you provide data for bind parameters by the `set`*xxx* 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.

This section includes the following topics:

- Using the setDataBuffer() Method
- Using the executeArrayUpdate() Method

Copyright © 1999, 2025, Oracle and/or its affiliates.

12.3.1 Using the 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:

```
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.

> **See Also**
>
> Table 5-2 in Data Types for specific cases

There is an important difference between the data provided by the `set`*xxx*`()` methods and `setDataBuffer()` method. When data is copied in the `set`*xxx*`()` 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 *i*th 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

12.3.2 Using the executeArrayUpdate() Method

If all data is provided with the `setDataBuffer()` methods or output streams (that is, no `set`*`xxx()`* 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 `set`*`xxx`*`()` 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:

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 12-2 How to Insert Records Using the addIteration() method

```
Statement *stmt = conn->createStatement(
  "insert into departments (department_id, department_name) values(:1, :2)");
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, OCCI_SQLT_STR, sizeof(dnames[0]), dnameLen);
stmt->addIteration();

stmt->setInt(1, 7654);        // specify data for the second row
                              // a setDatBuffer is unnecessary for the second
                              // bind parameter as data provided through
                              // setDataBuffer is specified only once.
stmt->executeUpdate();
```

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)");
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, OCCIINT, sizeof(ids[0]), idLen);
stmt->setDataBuffer(2, dnames, OCCI_SQLT_STR, sizeof(dnames[0]), dnameLen);

stmt->executeArrayUpdate(2);      // data for two rows is inserted.
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

12.4 Using the 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.

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.

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() );
```

12.5 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;
     }
     // take recovery action on the failed rows
   }
   ```

5. Catch and handle other errors generated during the statement update. Note that statement-level errors are still thrown as instances of a `SQLException`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
catch( SQLException &ex) // to catch other SQLExceptions.
{
   cout << "SQLException: " << e.what() << endl;
}
```

12.6 About Using Oracle Connection Manager in Traffic Director Mode

Oracle Connection Manager in Traffic Director Mode is a proxy that is placed between supported database clients and database instances.

Supported clients from Oracle Database 11*g* Release 2 (11.2) and later can connect to Oracle Connection Manager in Traffic Director Mode. Oracle Connection Manager in Traffic Director Mode provides improved high availability (HA) for planned and unplanned database server outages, connection multiplexing support, and load balancing. Support for Oracle Connection Manager in Traffic Director Mode is described in more detail in the following sections

- Modes of Operation
- Key Features

Modes of Operation

Oracle Connection Manager in Traffic Director Mode supports the following modes of operation:

- In pooled connection mode, Oracle Connection Manager in Traffic Director Mode supports any application using the following database client releases:
  - OCI, OCCI, and Open Source Drivers (Oracle Database 11*g* release 2 (11.2.0.4) and later))
  - JDBC (Oracle Database 12*c* release 1 (12.1) and later)
  - ODP.NET (Oracle Database 12*c* release 2 (12.2) and later)

  In addition, applications must use DRCP. That is, the application must enable DRCP in the connect string (or in the `tnsnames.ora` alias).
- In non-pooled connection (or dedicated) mode, Oracle Connection Manager in Traffic Director Mode supports any application using database client releases Oracle Database 11*g* release 2 (11.2.0.4) and later. In this mode, some capabilities, such as connection multiplexing are not available.

Key Features

Oracle Connection Manager in Traffic Director Mode furnishes support for the following:

- Transparent performance enhancements and connection multiplexing, which includes:
  - Statement caching, rows prefetching, and result set caching are auto-enabled for all modes of operation.
  - Database session multiplexing (pooled mode only) using the proxy resident connection pool (PRCP), where PRCP is a proxy mode of Database Resident Connection Pooling (DRCP). Applications get transparent connect-time load balancing and run-time load balancing between Oracle Connection Manager in Traffic Director Mode and the database.

Copyright © 1999, 2025, Oracle and/or its affiliates.

  - For multiple Oracle Connection Manager in Traffic Director Mode instances, applications get increased scalability through client-side connect time load balancing or with a load balancer (BIG-IP, NGINX, and others)

- Zero application downtime
  - Planned database maintenance or pluggable database (PDB) relocation
    * Pooled mode

      Oracle Connection Manager in Traffic Director Mode responds to Oracle Notification Service (ONS) events for planned outages and redirects work. Connections are drained from the pool on Oracle Connection Manager in Traffic Director Mode when the request completes. Service relocation is supported for Oracle Database 11*g* release 2 (11.2.0.4) and later.

      For PDB relocation, Oracle Connection Manager in Traffic Director Mode responds to in-band notifications when a PDB is relocated, that is even when ONS is not configured (for Oracle Database release 18c, version 18.1 and later server only)

    * Non-pooled or dedicated mode

      When there is no request boundary information from the client, Oracle Connection Manager in Traffic Director Mode supports planned outage for many applications (as long as only simple session state and cursor state need to be preserved across the request/transaction boundaries). This support includes:

      * Stop service/PDB at the transaction boundary or it leverages Oracle Database release 18c continuous application availability to stop the service at the request boundary
      * Oracle Connection Manager in Traffic Director Mode leverages Transparent Application Failover (TAF) failover restore to reconnect and restore simple states.
  - Unplanned database outages for read-mostly workloads
- High Availability of Oracle Connection Manager in Traffic Director Mode to avoid a single point of failure. This is supported by:
  - Multiple instances of Oracle Connection Manager in Traffic Director Mode using a load balancer or client side load balancing/failover in the connect string
  - Rolling upgrade of Oracle Connection Manager in Traffic Director Mode instances
  - Graceful close of existing connections from client to Oracle Connection Manager in Traffic Director Mode for planned outages
  - In-band notifications to Oracle Database release 18c and later clients
  - For older clients, notifications are sent with the response of the current request
- For security and isolation, Oracle Connection Manager in Traffic Director Mode furnishes:
  - Database Proxy supporting transmission control protocol/transmission control protocol secure (TCP/TCPS) and protocol conversion
  - Firewall based on the IP address, service name, and secure socket layer/transport layer security (SSL/TLS) wallets
  - Tenant isolation in a multi-tenant environment
  - Protection against denial-of-service and fuzzing attacks
  - Secure tunneling of database traffic across Oracle Database on-premises and Oracle Cloud

Copyright © 1999, 2025, Oracle and/or its affiliates.

See Also

- *Oracle AI Database Net Services Administrator's Guide* for information about configuring `cman.ora` configuration file to set up Oracle Connection Manager in Traffic Director Mode
- *Oracle AI Database Net Services Administrator's Guide* for information about configuring databases for Oracle Connection Manager in Traffic Director Mode proxy authentication
- *Oracle AI Database Net Services Administrator's Guide* for information about configuring Oracle Connection Manager in Traffic Director Mode for unplanned down events
- *Oracle AI Database Net Services Administrator's Guide* for information about configuring Oracle Connection Manager in Traffic Director Mode for planned down events
- *Oracle AI Database Net Services Administrator's Guide* for information about configuring proxy resident connection pools for use by Oracle Connection Manager in Traffic Director Mode
- *Oracle AI Database Net Services Administrator's Guide* for information about functionality not supported for all drivers with Oracle Connection Manager in Traffic Director Mode
- *Oracle AI Database Net Services Reference* for an overview of Oracle CMAN configuration file

12.7 About Run-time Load Balancing of the Stateless Connection Pool

Run-time load balancing in a stateless connection pool dynamically routs 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Oracle Call Interface Developer's Guide

This section includes the following topic: API Support.

12.7.1 API Support

New `NO_RLB` option for the `PoolType` attribute of the StatelessConnectionPool Class disables run-time load balancing.

12.8 About 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 AI Database Administrator's Guide*
>
> This section includes the following topics:
>
> - Using ADR Base Location
> - Using ADRCI
> - Controlling ADR Creation and Disabling Fault Diagnosability

12.8.1 Using 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 *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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- The *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 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 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 *$ORACLE_BASE* is /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 *$ORACLE_HOME* is /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
     ```

5. In the temporary directory.

Copyright © 1999, 2025, Oracle and/or its affiliates.

- 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 AI Database Net Services Reference*

12.8.2 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.

> **See Also**
>
> - *Oracle AI Database Utilities*for an introduction to the ADRCI

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
% adrci

ADRCI: Release 11.2. - on Wed November 25 16:16:55 2008

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/sttttd1/scratch/someone/view_storage/someone_v1/log
    set base .

adrci> set base /ade/chuck_l3/oracle/log

adrci> show incidents
...
adrci> quit
```

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:
*******************************************************************************
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
INCIDENT_ID    PROBLEM_KEY           CREATE_TIME
-------------------------------------------------------------------------
1                     oci 24550 [6]              2007-05-01 17:20:02.803697
-07:00
1 rows fetched

adrci> quit
```

12.8.3 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 Call Interface Developer's Guide

12.9 Using 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.

For usage guidelines, cache consistency, and restrictions, see Oracle Call Interface Developer's Guide.

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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
//cached on the client side.
rs = stmt->executeQuery();
```

12.10 About Client-Side Deployment Parameters and Auto Tuning

Starting with Oracle Database Release 12*c* Release 1 (12.1), Oracle provides `oraaccess.xml`, a client-side configuration file that can be used to configure selected properties, allowing the application behavior to be changed during deployment without modifying the source code.

> **Note**
>
> Do not use the prefetch deployment parameter if the OCCI application is already using the `setPrefetchRowCount()` or `setPrefetchMemorySize()` methods.

> **See**
>
> Oracle Call Interface Developer's Guide for more information about client-side deployment parameters and auto tuning

12.10.1 Support for Centralized Configuration Store

This section discusses Oracle C++ Call Interface (OCCI) support for Centralized Configuration Store.

Centralized Configuration Store can be Azure-based or Oracle Cloud Infrastructure based. Centralized Configuration Store manages all application configurations centrally.

> **See Also**
>
> - createConnection()
> - Overview of Centralized Configuration Provider Naming
> - OCI Support for Centralised Configuration Store

12.10.1.1 Examples for Centralized Configuration Store OCCI Support

Example 12-9 Standalone Connection

In this example, the user provides the username and password externally. If the username and password are stored in the Centralized Configuration Store (Config Store), then they are not effective because of the precedence order of credentials.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> db is the Connect String to the Centralized Configuration Store.

```
env = Environment::createEnvironment(Environment::DEFAULT);
conn = env->createConnection(user,password, db);

In this case, username, password and db are in UTF16 format
env = Environment::createEnvironment("OCCIUTF16", "OCCIUTF16", Environment::
DEFAULT);
conn = env->createConnection(username, password, db);  // This case is UTF16
version of the above API.

In this case,User credentials are picked from the Config Store and used for
database connection.
env = Environment::createEnvironment(Environment::DEFAULT);
conn = env->createConnection(db);

In this case, db is in UTF16 format
env = Environment::createEnvironment("OCCIUTF16", "OCCIUTF16", Environment::
DEFAULT);
conn = env->createConnection(db);  // This case is UTF16 version of the above
API.

In this case, ConnectString is expected to be set in TWO_TASK, and it should
point to the Config Store to use the TWO_TASK
environment variable.
env = Environment::createEnvironment(Environment::DEFAULT);
conn = env->createConnection();
```

Example 12-10 Session Pool

In this example, connectString is the URL to the Centralized Configuration Store and all the pool parameters are overridden by the parameter values stored in Centralized Configuration Store.

```
env = Environment::createEnvironment (Environment::DEFAULT);
StatelessConnectionPool *scPool;
Connection *con1, *con2, *con3;
Statement *stmt1, *stmt2, *stmt3;

// Create a homogenous connection pool
scPool = env->createStatelessConnectionPool
               (poolUserName, poolPassword, connectString, maxConn,
                minConn, incrConn, StatelessConnectionPool::HOMOGENEOUS);
con1 = scPool->getConnection();
stmt1 = con1->createStatement(sqlquery1);
ResultSet *rset1 = stmt1->executeQuery();
cout<<"Retrieving the data"<<endl;
while (rset1->next())
{
    cout<<"Emp id: "<<rset1->getInt(1)<<endl;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
        }
        stmt1->closeResultSet(rset1);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13
OCCI Application Programming Interface

This chapter describes the OCCI classes and methods for C++.

> **See Also**
>
> - Format Models in *Oracle AI Database SQL Language Reference*
> - Table A-1 in *Oracle AI Database Globalization Support Guide*

13.1 OCCI Classes and Methods

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

Class	Description
Agent Class	Represents an agent in the Advanced Queuing context.
AnyData Class	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.
BatchSQLException Class	Provides methods for handling batch processing errors; extends the SQLException Class.
Bfile Class	Provides access to a SQL `BFILE` value.
Blob Class	Provides access to a SQL `BLOB` value.
Bytes Class	Examines individual bytes of a sequence for comparing bytes, searching bytes, and extracting bytes.
Clob Class	Provides access to a SQL `CLOB` value.
Connection Class	Represents a connection with a specific database.
ConnectionPool Class	Represents a connection pool with a specific database.
Consumer Class	Supports dequeuing of `Messages` and controls the dequeuing options.
Date Class	Specifies abstraction for SQL DATE data items. Also provides formatting and parsing operations to support the OCCI escape syntax for date values.
Environment Class	Provides an OCCI environment to manager memory and other resources of OCCI objects. An OCCI driver manager maps to an OCCI environment handle.
IntervalDS Class	Represents a time period in terms of days, hours, minutes, and seconds.
IntervalYM Class	Represents a time period in terms of year and months.
Listener Class	Listens on behalf of one or more agents on one or more queues.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-1 (Cont.) Summary of OCCI Classes

Class	Description
Map Class	Used to store the mapping of the SQL structured type to C++ classes.
Message Class	A unit that is enqueued or dequeued.
MetaData Class	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.
NotifyResult Class	Used to hold notification information from the Database AQ callback function.
Number Class	Models the numeric data type.
PObject Class	When defining types, enables specification of persistent or transient instances. Class instances derived from `PObject` can be either persistent or transient. If persistent, a class instance derived from `PObject` inherits from the `PObject` class; if transient, there is no inheritance.
Producer Class	Supports enqueuing options and enqueues `Messages`.
Ref Class	The mapping in C++ for the SQL REF value, which is a reference to a SQL structured type value in the database.
RefAny Class	The mapping in C++ for the SQL REF value, which is a reference to a SQL structured type value in the database.
ResultSet Class	Provides access to a table of data generated by executing an OCCI `Statement`.
SQLException Class	Provides information on database access errors.
StatelessConnectionPool Class	Represents a pool of stateless, authenticated connections to the database.
Statement Class	Used for executing SQL statements, including both query statements and insert / update / delete statements.
Stream Class	Used to provide streamed data (usually of the LONG data type) to a prepared DML statement or stored procedure call.
Subscription Class	Encapsulates the information and operations necessary for registering a subscriber for notification.
Timestamp Class	Specifies abstraction for SQL TIMESTAMP data items. Also provides formatting and parsing operations to support the OCCI escape syntax for time stamp values.

13.1.1 Using OCCI Classes

OCCI classes are defined in the `oracle::occi` namespace. An OCCI class name within the `oracle::occi` 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Using Scope Resolution Operator for OCCI

The scope resolution operator (`::`) is used to explicitly specify the `oracle::occi` namespace and the OCCI class name. To declare `myConnection`, a `Connection` object, using the scope resolution operator, you would use the following syntax:

```
oracle::occi::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::occi` namespace, you would use the following syntax:

```
using oracle::occi::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.

13.1.2 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`.

This section includes the following topic: Working with Collections of Refs.

13.1.2.1 Working with Collections of Refs

Collections of Refs can be fetched and inserted using methods of the following classes:

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.1.2.1.1 ResultSet Class

Fetching Collection of Refs

Use the following version of getVectorOfRefs() to return a column of references:

```
void getVectorOfRefs(
   ResultSet  *rs,
   unsigned int index,
   vector<Ref<T> > &vect);
```

13.1.2.1.2 Statement Class

Fetching Collection of Refs

Use getVectorOfRefs() to return a collection of references from a column:

```
void getVectorOfRefs(
   Statement  *stmt,
   unsigned int index,
   vector<Ref<T> > &vect);
```

Inserting a Collection of Refs

Use setVectorOfRefs() to insert a collection of references into a column:

```
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:

```
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:

```
template <class T>
void setVectorOfRefs(
   Statement *stmt,
   unsigned int paramIndex,
   const vector<Ref<T> > &vect,
   const UString &schemaName,
   const UString &typeName);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.2 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

Attribute	Options
`LockOptions`	• `OCCI_LOCK_NONE` clears the lock setting on the `Ref` object. • `OCCI_LOCK_X` indicates that the object should be locked, and to wait for the lock to be available if the object is locked by another session. • `OCCI_LOCK_X_NOWAIT` indicates that the object should be locked, and returns an error if it is locked by another session.
`CharSetForm`	• `OCCI_SQLCS_IMPLICIT` indicates that the local database character set must be used. • `OCCI_SQLCS_NCHAR` indicates that the local database `NCHAR` set must be used. • `OCCI_SQLCS_EXPLICIT` indicates that the character set is specified explicitly. • `OCCI_SQLCS_FLEXIBLE` means that the character set is a PL/SQL flexible parameter.
`ReturnStatus`	• `OCCI_SUCCESS` indicates that the call has been made successfully (transaction failover mode). • `FO_RETRY` indicates that the call should be retried (transaction failover mode).

13.3 Agent Class

The `Agent` class represents an agent in the Advanced Queuing context.

Table 13-3 Summary of Agent Methods

Method	Summary
Agent()	`Agent` class constructor.
getAddress()	Returns the address of the `Agent`.
getName()	Returns the name of the `Agent`.
getProtocol()	Returns the protocol of the `Agent`.
isNull()	Tests whether the `Agent` object is `NULL`.
operator=()	Assignment operator for `Agent`.
setAddress()	Sets address of the `Agent` object.
setName()	Sets name of the `Agent` object.
setNull()	Sets `Agent` object to `NULL`.
setProtocol()	Sets protocol of the `Agent` object.

13.3.1 Agent()

Agent class constructor.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`Agent( const Environment *env);`	Creates an `Agent` object initialized to its default values.
`Agent( const Agent& agent);`	Copy constructor.
`Agent( const Environment *env, const string& name, const string& address, unsigned int protocol = 0);`	Creates an `Agent` object with specified `Agent`'s name, address, and protocol.

Parameter	Description
`env`	Environment
`name`	Name
`agent`	Original agent
`address`	Address
`protocol`	Protocol

13.3.2 getAddress()

Returns a string containing `Agent`'s address.

Syntax

```
string getAddress() const;
```

13.3.3 getName()

Returns a string containing `Agent`'s name.

Syntax

```
string getName() const;
```

13.3.4 getProtocol()

Returns a numeric code representing `Agent`'s protocol.

Syntax

```
unsigned int getProtocol() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.3.5 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;
```

13.3.6 operator=()

Assignment operator for `Agent` class.

Syntax

```
void operator=(
const Agent& agent);
```

Parameter	Description
`agent`	The original `Agent` object.

13.3.7 setAddress()

Sets the address of the `Agent` object.

Syntax

```
void setAddress(
   const string& addr);
```

Parameter	Description
`addr`	The name of the `Agent` object.

13.3.8 setName()

Sets the name of the `Agent` object.

Syntax

```
void setName(
   const string& name);
```

Parameter	Description
`name`	The name of the `Agent` object.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.3.9 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

```
void setNull();
```

13.3.10 setProtocol()

Sets the protocol of the `Agent` object.

Syntax

```
void setProtocol(
   unsigned int protocol = 0);
```

Parameter	Description
`protocol`	The protocol of the `Agent` object.

13.4 AnyData Class

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 `setFrom`*`xxx`*`()` methods. An `AnyData` object can be converted into most SQL and user-defined types using `getAs`*`xxx`*`()` methods. `SYS.ANYDATA` type models `AnyData` both in SQL and PL/SQL. See Table 13-4 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`.

```
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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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.

```
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( "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 13-4 OCCI Data Types supported by AnyData Class

Data Type	TypeCode
BDouble	OCCI_TYPECODE_BDOUBLE

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-4 (Cont.) OCCI Data Types supported by AnyData Class

Data Type	TypeCode
BFile	OCCI_TYPECODE_BFILE
BFloat	OCCI_TYPECODE_BFLOAT
Bytes	OCCI_TYPECODE_RAW
Date	OCCI_TYPECODE_DATE
IntervalDS	OCCI_TYPECODE_INTERVAL_DS
IntervalYM	OCCI_TYPECODE_INTERVAL_YM
Number	OCCI_TYPECODE_NUMBERB
PObject	OCCI_TYPECODE_OBJECT
Ref	OCCI_TYPECODE_REF
string	OCCI_TYPECODE_VARCHAR2
TimeStamp	OCCI_TYPECODE_TIMESTAMP

Table 13-5 Summary of AnyData Methods

Method	Summary
AnyData()	AnyData class constructor.
getAsBDouble()	Converts an AnyData object into BDouble.
getAsBfile()	Converts an AnyData object into Bfile.
getAsBFloat()	Converts an AnyData object into BFloat.
getAsBytes()	Converts an AnyData object into Bytes.
getAsDate()	Converts an AnyData object into Date.
getAsIntervalDS()	Converts an AnyData object into IntervalDS.
getAsIntervalYM()	Converts an AnyData object into IntervalYM.
getAsNumber()	Converts an AnyData object into Number.
getAsObject()	Converts an AnyData object into PObject.
getAsRef()	Converts an AnyData object into RefAny.
getAsString()	Converts an AnyData object into a namespace string.
getAsTimestamp()	Converts an AnyData object into Timestamp.
getType()	Retrieves the DataType held by the AnyData object. See Table 13-4.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-5 (Cont.) Summary of AnyData Methods

Method	Summary
isNull()	Tests whether AnyData object is NULL.
setFromBDouble()	Converts a BDouble into Anydata.
setFromBfile()	Converts a Bfile into Anydata.
setFromBFloat()	Converts a BFloat into Anydata.
setFromBytes()	Converts a Bytes into Anydata.
setFromDate()	Converts a Date into Anydata.
setFromIntervalDS()	Converts an IntervalDS into Anydata.
setFromIntervalYM()	Converts an IntervalYM into Anydata.
setFromNumber()	Converts a Number into Anydata.
setFromObject()	Converts a PObject into Anydata.
setFromRef()	Converts a RefAny into Anydata.
setFromString()	Converts a namespace string into Anydata.
setFromTimestamp()	Converts a Timestamp into Anydata.
setNull()	Sets AnyData object to NULL.

13.4.1 AnyData()

AnyData constructor.

Syntax

```
AnyData(
   const Connection *conn);
```

Parameter	Description
conn	The connection.

13.4.2 getAsBDouble()

Converts an AnyData object into BDouble.

Syntax

```
BDouble getAsBDouble() const;
```

13.4.3 getAsBfile()

Converts an AnyData object into Bfile.

Syntax

```
Bfile getAsBfile() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.4.4 getAsBFloat()

Converts an `AnyData` object into `BFloat`.

Syntax

```
BFloat getAsBFloat() const;
```

13.4.5 getAsBytes()

Converts an `AnyData` object into Bytes.

Syntax

```
Bytes getAsBytes() const;
```

13.4.6 getAsDate()

Converts an `AnyData` object into `Date`.

Syntax

```
Date getAsDate() const;
```

13.4.7 getAsIntervalDS()

Converts an `AnyData` object into `IntervalDS`.

Syntax

```
IntervalDS getAsIntervalDS() const;
```

13.4.8 getAsIntervalYM()

Converts an `AnyData` object into `IntervalYM`.

Syntax

```
IntervalYS getAsIntervalYM() const;
```

13.4.9 getAsNumber()

Converts an `AnyData` object into `Number`.

Syntax

```
Number getAsNumber() const;
```

13.4.10 getAsObject()

Converts an `AnyData` object into `PObject`.

Syntax

```
PObject* getAsObject() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.4.11 getAsRef()

Converts an `AnyData` object into `RefAny`.

Syntax

```
RefAny getAsRef() const;
```

13.4.12 getAsString()

Converts an `AnyData` object into a namespace `string`.

Syntax

```
string getAsString() const;
```

13.4.13 getAsTimestamp()

Converts an `AnyData` object into `Timestamp`.

Syntax

```
Timestamp getAsTimestamp() const;
```

13.4.14 getType()

Retrieves the data type held by the `AnyData` object. Refer to Table 13-4 for valid values for `TypeCode`.

Syntax

```
TypeCode getType();
```

13.4.15 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;
```

13.4.16 setFromBDouble()

Converts a `BDouble` into `AnyData`.

Syntax

```
void setFromBDouble(
   const BDouble& bdouble);
```

Parameter	Description
`bdouble`	The `BDouble` that is converted into `AnyData`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.4.17 setFromBfile()

Converts a `Bfile` into `AnyData`.

Syntax

```
void setFromBfile(
   const Bfile& bfile);
```

Parameter	Description
`bfile`	The `Bfile` that is converted into `AnyData`.

13.4.18 setFromBFloat()

Converts a `BFloat` into `AnyData`.

Syntax

```
void setFromBFloat(
   const BFloat& bfloat);
```

Parameter	Description
`bfloat`	The `BFloat` that is converted into `AnyData`.

13.4.19 setFromBytes()

Converts a `Bytes` into `AnyData`.

Syntax

```
void setFromBytes(
   const Bytes& bytes);
```

Parameter	Description
`bytes`	The `Bytes` that is converted into `AnyData`.

13.4.20 setFromDate()

Converts a `Date` into `AnyData`.

Syntax

```
void setFromDate(
   const Date& date);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
date	The Date that is converted into AnyData.

13.4.21 setFromIntervalDS()

Converts an IntervalDS into AnyData.

Syntax

```
void setFromIntervalDS(
   const IntervalDS& intervalds);
```

Parameter	Description
invtervalds	The IntervalDS that is converted into AnyData.

13.4.22 setFromIntervalYM()

Converts an IntervalYM into AnyData.

Syntax

```
void setFromIntervalYM(
   const IntervalYM& intervalym);
```

Parameter	Description
invalym	The IntervalYM that is converted into AnyData.

13.4.23 setFromNumber()

Converts a Number into AnyData.

Syntax

```
void setFromNumber(
   const Number& num);
```

Parameter	Description
num	The Number that is converted into AnyData.

13.4.24 setFromObject()

Converts a PObject into AnyData.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setFromObject(
   const PObject* objptr);
```

Parameter	Description
objptr	The PObject that is converted into AnyData.

13.4.25 setFromRef()

Converts a PObject into AnyData.

Syntax

```
void setFromRef(
   const RefAny& ref
   const string &typeName,
   const string &schema);
```

Parameter	Description
ref	The RefAny that is converted into AnyData.
typeName	The name of the type.
schema	The name of the schema where the type is defined.

13.4.26 setFromString()

Converts a namespace string into AnyData.

Syntax

```
void setFromString(
   string& str);
```

Parameter	Description
str	The namespace string that is converted into AnyData.

13.4.27 setFromTimestamp()

Converts a Timestamp into AnyData.

Syntax

```
void setFromTimestamp(
   const Timestamp& timestamp);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
`timestamp`	The `Timestamp` that is converted into `AnyData`.

13.4.28 setNull()

Sets `AnyData object` to `NULL`.

Syntax

```
void setNull();
```

13.5 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 Optimizing Performance of C++ Applications.

Table 13-6 Summary of BatchSQLException Methods

Method	Summary
getException()	Returns the exception.
getFailedRowCount()	Returns the number of rows with failed inserts or updates.
getRowNum()	Returns the number of the row that has an insert or updated error

13.5.1 getException()

Returns the exception that matches the specified `index`.

Syntax

```
SQLException getSQLException (
   unsigned int index) const;
```

Parameter	Description
`index`	The index into the list of errors returned by the batch process.

13.5.2 getFailedRowCount()

Returns the number of rows for which the statement insert or update failed.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
unsigned int getFailedRowCount( ) const;
```

13.5.3 getRowNum()

Returns the number of the row with an error, matching the specified `index`.

Syntax

```
unsigned int getRowNum(
   unsigned int index) const;
```

Parameter	Description
`index`	The index into the list of errors returned by the batch process.

13.6 Bfile Class

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 Oracle AI 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 AI Database SecureFiles and Large Objects Developer's Guide*,

Table 13-7 Summary of Bfile Methods

Method	Summary
Bfile()	`Bfile` class constructor.
close()	Closes a previously opened `BFILE`.
closeStream()	Closes the stream obtained from the `BFILE`.
fileExists()	Tests whether the `BFILE` exists.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-7 (Cont.) Summary of Bfile Methods

Method	Summary
getDirAlias()	Returns the directory object of the BFILE.
getFileName()	Returns the name of the BFILE.
getStream()	Returns data from the BFILE as a Stream object.
getUStringDirAlias()	Returns a UString containing the directory object associated with the BFILE.
getUStringFileName()	Returns a UString containing the file name associated with the BFILE.
isInitialized()	Tests whether the Bfile object is initialized.
isNull()	Tests whether the Bfile object is atomically NULL.
isOpen()	Tests whether the BFILE is open.
length()	Returns the number of bytes in the BFILE.
open()	Opens the BFILE with read-only access.
operator=()	Assigns a BFILE locator to the Bfile object.
operator==()	Tests whether two Bfile objects are equal.
operator!=()	Tests whether two Bfile objects are not equal.
operator==()	Reads a specified portion of the BFILE into a buffer.
setName()	Sets the directory object and file name of the BFILE.
setNull()	Sets the Bfile object to atomically NULL.

13.6.1 Bfile()

Bfile class constructor.

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.6.2 close()

Closes a previously opened `Bfile`.

Syntax

```
void close();
```

13.6.3 closeStream()

Closes the stream obtained from the `Bfile`.

Syntax

```
void closeStream(
   Stream *stream);
```

Parameter	Description
`stream`	The stream to be closed.

13.6.4 fileExists()

Tests whether the `BFILE` exists. If the `BFILE` exists, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool fileExists() const;
```

13.6.5 getDirAlias()

Returns a string containing the directory object associated with the `BFILE`.

Syntax

```
string getDirAlias() const;
```

13.6.6 getFileName()

Returns a string containing the file name associated with the `BFILE`.

Syntax

```
string getFileName() const;
```

13.6.7 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
Stream* getStream(
   unsigned int offset = 1,
   unsigned int amount = 0);
```

Parameter	Description
`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 `BLOB`. Valid values are numbers greater than or equal to 1.
`amount`	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`.

13.6.8 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

```
UString getUStringDirAlias() const;
```

13.6.9 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;
```

13.6.10 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;
```

13.6.11 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.6.12 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;
```

13.6.13 length()

Returns the number of bytes (inclusive of the end of file marker) in the `BFILE`.

Syntax

```
unsigned int length() const;
```

13.6.14 open()

Opens an existing `BFILE` for read-only access. This function is meaningful the first time it is called for a `Bfile` object.

Syntax

```
void open();
```

13.6.15 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

```
Bfile& operator=(
   const Bfile &srcBfile);
```

Parameter	Description
`srcBfile`	The `Bfile` object to be assigned to the current `Bfile` object.

13.6.16 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

```
bool operator==(
   const Bfile &srcBfile) const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
srcBfile	The Bfile object to be compared with the current Bfile object.

13.6.17 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

```
bool operator!=(
   const Bfile &srcBfile) const;
```

Parameter	Description
srcBfile	The Bfile object to be compared with the current Bfile object.

13.6.18 read()

Reads a part or all of the BFILE into the buffer specified, and returns the number of bytes read.

Syntax

```
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.
buffsize	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.

13.6.19 setName()

Sets the directory object and file name of the BFILE.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void setName(`
`  const string &dirAlias,`
`  const string &fileName);`
	Sets the directory object and file name of the `BFILE`.

`void setName(`
`  const UString &dirAlias,`
`  const UString &fileName);`
	Sets the directory object and file name of the `BFILE` (Unicode support). The client `Environment` should be initialized in OCCIUTIF16 mode.

Parameter	Description
`dirAlias`	The directory object to be associated with the `BFILE`.
`fileName`	The file name to be associated with the `BFILE`.

13.6.20 setNull()

Sets the `Bfile object` to atomically `NULL`.

Syntax

```
void setNull();
```

13.7 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 AI Database SecureFiles and Large Objects Developer's Guide*

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-8 Summary of Blob Methods

Method	Summary
Blob()	`Blob` class constructor.
append()	Appends a specified `BLOB` to the end of the current `BLOB`.
close()	Closes a previously opened `BLOB`.
closeStream()	Closes the `Stream` object obtained from the `BLOB`.
copy()	Copies a specified portion of a `BFILE` or `BLOB` into the current `BLOB`.
getChunkSize()	Returns the smallest data size to perform efficient writes to the `BLOB`.
getContentType()	Returns the content type of the `Blob`.
getOptions()	Returns the `BLOB`'s `LobOptionValue` for a specified `LobOptionType`.
getStream()	Returns data from the `BLOB` as a `Stream` object.
isInitialized()	Tests whether the `Blob` object is initialized
isNull()	Tests whether the `Blob` object is atomically `NULL`.
isOpen()	Tests whether the `BLOB` is open.
length()	Returns the number of bytes in the `BLOB`.
open()	Opens the `BLOB with read or` read/write access.
operator=()	Assigns a `BLOB` locator to the `Blob` object.
operator==()	Tests whether two `Blob` objects are equal.
operator!= ()	Tests whether two `Blob` objects are not equal.
read()	Reads a portion of the `BLOB` into a buffer.
setContentType()	Sets the content type of the `Blob`.
setEmpty()	Sets the `Blob` object to empty.
setNull()	Sets the `Blob` object to atomically `NULL`.
setOptions()	Specifies a `LobOptionValue` for a particular `LobOptionType`. Enables advanced compression, encryption and deduplication of `BLOB`s.
trim()	Truncates the `BLOB` to a specified length.
write()	Writes a buffer into an *unopened* `BLOB`.
writeChunk()	Writes a buffer into an *open* `BLOB`.

13.7.1 Blob()

`Blob` class constructor.

Syntax	Description
`Blob();`	Creates a `NULL Blob` object.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`Blob( const Connection *connectionp);`	Creates an uninitialized `Blob` object.
`Blob( const Blob &srcBlob);`	Creates a copy of a `Blob` object.

Parameter	Description
`connectionp`	The connection pointer
`srcBlob`	The source `Blob` object.

13.7.2 append()

Appends a `BLOB` to the end of the current `BLOB`.

Syntax

```
void append(
   const Blob &srcBlob);
```

Parameter	Description
`srcBlob`	The `BLOB` object to be appended to the current `BLOB` object.

13.7.3 close()

Closes a `BLOB`.

Syntax

```
void close();
```

13.7.4 closeStream()

Closes the Stream object obtained from the `BLOB`.

Syntax

```
void closeStream(
   Stream *stream);
```

Parameter	Description
`stream`	The `Stream` to be closed.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.7.5 copy()

Copies a part or all of a BFILE or BLOB into the current BLOB.

Syntax	Description

`void copy(`
`  const Bfile &srcBfile,`
`  unsigned int numBytes,`
`  unsigned int dstOffset = 1,`
`  unsigned int srcOffset = 1);`
	Copies a part of a BFILE into the current BLOB.

`void copy(`
`  const Blob &srcBlob,`
`  unsigned int numBytes,`
`  unsigned int dstOffset = 1,`
`  unsigned int srcOffset = 1);`
	Copies a part of a BLOB into the current BLOB.
	If the destination BLOB has deduplication enabled, and the source and destination BLOBs are in the same column, the new BLOB is created as copy-on-write. All other settings are inherited from the source BLOB. If the destination BLOB has deduplication disabled, it is a completely new copy of the BLOB.

Parameter	Description
srcBfile	The BFILE from which the data is to be copied.
srcBlob	The BLOB from which the data is to be copied.
numBytes	The number of bytes to be copied from the source BFILE or BLOB. Valid values are numbers greater than 0.
dstOffset	The starting position at which to begin writing data into the current BLOB. Valid values are numbers greater than or equal to 1.
srcOffset	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.

13.7.6 getChunkSize()

Returns the smallest data size to perform efficient writes to the BLOB.

Syntax

```
unsigned int getChunkSize() const;
```

13.7.7 getContentType()

Returns the content type of the Blob. If a content type has not been assigned, returns a NULL string.

Syntax

```
string getContentType();
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.7.8 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);
```

Parameter	Description
optType	The LobOptionType setting requested. These may be combined using bitwise or (|) to avoid server round trips. See Table 7-1 and Table 7-2

13.7.9 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);
```

Parameter	Description
offset	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.
amount	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.

13.7.10 isInitialized()

Tests whether the Blob object is initialized. If the Blob object is initialized, then TRUE is returned; otherwise, FALSE is returned.

Syntax

```
bool isInitialized() const;
```

13.7.11 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

```
bool isNull() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.7.12 isOpen()

Tests whether the BLOB is open. If the BLOB is open, then TRUE is returned; otherwise, FALSE is returned.

Syntax

```
bool isOpen() const;
```

13.7.13 length()

Returns the number of bytes in the BLOB.

Syntax

```
unsigned int length() const;
```

13.7.14 open()

Opens the BLOB in read/write or read-only mode.

Syntax

```
void open(
   LobOpenMode mode = OCCI_LOB_READWRITE);
```

Parameter	Description
mode	The mode the BLOB is to be opened in. Valid values are: • OCCI_LOB_READWRITE • OCCI_LOB_READONLY

13.7.15 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

```
Blob& operator=(
   const Blob &srcBlob);
```

Parameter	Description
srcBlob	The source BLOB from which to copy data.

13.7.16 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool operator==(
   const Blob &srcBlob) const;
```

Parameter	Description
srcBlob	The source BLOB to be compared with the current BLOB.

13.7.17 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

```
bool operator!=(
   const Blob &srcBlob) const;
```

Parameter	Description
srcBlob	The source BLOB to be compared with the current BLOB.

13.7.18 read()

Reads a part or all of the BLOB into a buffer. The actual number of bytes read is returned.

Syntax

```
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 BLOB data is to be read into. Valid values are numbers greater than or equal to amt.
buffsize	The size of the buffer that the BLOB 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 BLOB. If offset is not specified, the data is written from the beginning of the BLOB.

13.7.19 setContentType()

Sets the content type of the Blob. If the Blob is not a SecureFile, throws an exception.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setContentType(
   const string contenttype);
```

Parameter	Description
`contenttype`	The content type of the `Blob`; an ASCII Mime compliant string.

13.7.20 setEmpty()

Sets the `Blob` object to empty.

Syntax	Description
`void setEmpty();`	Sets the `Blob` object to empty.
`void setEmpty(` `  const Connection* connectionp);`	Sets the `Blob` object to empty and initializes the connection pointer to the passed parameter.

Parameter	Description
`connectionp`	The new connection pointer for the `BLOB` object.

13.7.21 setNull()

Sets the `Blob` object to atomically `NULL`.

Syntax

```
void setNull();
```

13.7.22 setOptions()

Specifies a `LobOptionValue` for a particular `LobOptionType`. Enables advanced compression, encryption and deduplication of `BLOB`s. See Table 7-1 and Table 7-2.

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

```
void setOptions(
   LobOptionType optType,
   LobOptionValue optValue);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
optType	The LobOptionType setting being specified. These may be combined using bitwise or (|) to avoid server round trips.
optValue	The LobOptionValue setting for the LobOptionType specified by the optType parameter

13.7.23 trim()

Truncates the BLOB to the new length specified.

Syntax

```
void trim(
   unsigned int newlen);
```

Parameter	Description
newlen	The new length of the BLOB. Valid values are numbers less than or equal to the current length of the BLOB.

13.7.24 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

```
unsigned int write(
   unsigned int amt,
   unsigned char *buffer,
   unsigned int bufsize,
   unsigned int offset = 1);
```

Parameter	Description
amt	The number of bytes to be written to the BLOB.
buffer	The buffer containing the data to be written to the BLOB.
buffsize	The size of the buffer containing the data to be written to the BLOB. Valid values are numbers greater than or equal to *amt*.
offset	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.

13.7.25 writeChunk()

Writes data from a buffer into a previously opened BLOB. The actual number of bytes written is returned.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
unsigned int writeChunk(
   unsigned int amount,
   unsigned char *buffer,
   unsigned int bufsize,
   unsigned int offset = 1);
```

Parameter	Description
`amt`	The number of bytes to be written to the `BLOB`.
`buffer`	The buffer containing the data to be written to the `BLOB`.
`buffsize`	The size of the buffer containing the data to be written to the `BLOB`. Valid values are numbers greater than or equal to *`amt`*.
`offset`	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`.

13.8 Bytes Class

Methods of the `Bytes` class enable you to perform specific tasks related to `Bytes` objects.

Table 13-9 Summary of Bytes Methods

Method	Summary
Bytes()	`Bytes` class constructor.
byteAt()	Returns the byte at the specified position of the `Bytes` object.
getBytes()	Returns a byte array from the `Bytes` object.
isNull()	Tests whether the `Bytes` object is `NULL`.
length()	Returns the number of bytes in the `Bytes` object.
operator=()	Assignment operator for `Bytes` class.
setNull()	Sets the `Bytes` object to `NULL`.

13.8.1 Bytes()

`Bytes` class constructor.

Syntax	Description
`Bytes(` `  Environment *env = NULL);`	Creates a `Bytes` object.

`Bytes(`
`  unsigned char *value,`
`  unsigned int count`
`  unsigned int offset = 0,`
`  const Environment *env = NULL);`
	Creates a `Bytes` object that contains a subarray of bytes from a character array.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`Bytes(` `  const Bytes &e);`	Creates a copy of a `Bytes` object, use the syntax

Parameter	Description
`env`	Environment
`value`	Initial value of the new object
`count`	The size of the subset of the character array that is copied into the new bytes object
`offset`	The first position from which to begin copying the character array
`e`	The source `Bytes` object.

13.8.2 byteAt()

Returns the byte at the specified position in the `Bytes` object.

Syntax

```
unsigned char byteAt(
   unsigned int index) const;
```

Parameter	Description
`index`	The position of the byte to be returned from the `Bytes` object; the first byte of the `Bytes` object is at 0.

13.8.3 getBytes()

Copies 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;
```

Parameter	Description
`dst`	The destination buffer into which data from the `Bytes` object is to be written.
`count`	The number of bytes to copy.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
srcBegin	The starting position at which data is to be read from the Bytes object; the position of the first byte in the Bytes object is at 0.
dstBegin	The starting position at which data is to be written in the destination buffer; the position of the first byte in dst is at 0.

13.8.4 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;
```

13.8.5 length()

This method returns the length of the Bytes object.

Syntax

```
unsigned int length() const;
```

13.8.6 operator=()

Assignment operator for Bytes class.

Syntax

```
void operator=(
   const Bytes& bytes);
```

Parameter	Description
bytes	The original Bytes.

13.8.7 setNull()

This method sets the Bytes object to atomically NULL.

Syntax

```
void setNull();
```

13.9 Clob Class

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 AI Database SecureFiles and Large Objects Developer's Guide*,

Table 13-10 Summary of Clob Methods

Method	Summary
Clob()	Clob class constructor.
append()	Appends a Clob at the end of the current Clob.
close()	Closes a previously opened Clob.
closeStream()	Closes the Stream object obtained from the current Clob.
copy()	Copies all or a portion of a Clob or BFILE into the current Clob.
getCharSetForm()	Returns the character set form of the Clob.
getCharSetId()	Returns the character set ID of the Clob.
getCharSetIdUString()	Retrieves the characterset name associated with the Clob; UString version.
getChunkSize()	Returns the smallest data size to perform efficient writes to the CLOB.
getContentType()	Returns the content type of the Clob.
getOptions()	Returns the CLOB's LobOptionValue for a specified LobOptionType.
getStream()	Returns data from the CLOB as a Stream object.
isInitialized()	Tests whether the Clob object is initialized.
isNull()	Tests whether the Clob object is atomically NULL.
isOpen()	Tests whether the Clob is open.
length()	Returns the number of characters in the current CLOB.
open()	Opens the CLOB with read or read/write access.
operator=()	Assigns a CLOB locator to the current Clob object.
operator==()	Tests whether two Clob objects are equal.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-10 (Cont.) Summary of Clob Methods

Method	Summary
operator!=()	Tests whether two `Clob` objects are not equal.
read()	Reads a portion of the `CLOB` into a buffer.
setCharSetId()	Sets the character set ID associated with the `Clob`.
setCharSetIdUString()	Sets the character set ID associated with the `Clob`; used when the environment character set is UTF16.
setCharSetForm()	Sets the character set form associated with the `Clob`.
setContentType()	Sets the content type of the `Clob`.
setEmpty()	Sets the `Clob` object to empty.
setNull()	Sets the `Clob` object to atomically `NULL`.
setOptions()	Specifies a `LobOptionValue` for a particular `LobOptionType`. Enables advanced compression, encryption and deduplication of `CLOB`s.
trim()	Truncates the `Clob` to a specified length.
write()	Writes a buffer into an *unopened* `CLOB`.
writeChunk()	Writes a buffer into an *open* `CLOB`.

13.9.1 Clob()

`Clob` class constructor.

Syntax	Description
`Clob();`	Creates a `NULL` `Clob` object.
`Clob(` `  const Connection *connectionp);`	Creates an uninitialized `Clob` object.
`Clob(` `  const Clob *srcClob);`	Creates a copy of a `Clob` object.

Parameter	Description
`connectionp`	Connection pointer
`srcClob`	The source `Clob` object

13.9.2 append()

Appends a `CLOB` to the end of the current `CLOB`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void append(
   const Clob &srcClob);
```

Parameter	Description
srcClob	The CLOB to be appended to the current CLOB.

13.9.3 close()

Closes a CLOB.

Syntax

```
void close();
```

13.9.4 closeStream()

Closes the Stream object obtained from the CLOB.

Syntax

```
void closeStream(
   Stream *stream);
```

Parameter	Description
stream	The Stream object to be closed.

13.9.5 copy()

Copies a part or all of a BFILE or CLOB into the current CLOB.

OCCI does not perform any characterset 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 characterset.

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`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.

Parameter	Description
srcBfile	The BFILE from which the data is to be copied.
srcClob	The CLOB from which the data is to be copied.
numBytes	The number of bytes to be copied from the source BFILE or CLOB. Valid values are numbers greater than 0.
dstOffset	The starting position at which data is to be is at 0. The starting position at which to begin writing data into the current CLOB Valid values are numbers greater than or equal to 1 written in the destination buffer; the position of the first byte.
srcOffset	The starting position at which to begin reading data from the source BFILE or CLOB. Valid values are numbers greater than or equal to 1.

13.9.6 getCharSetForm()

Returns the character set form of the CLOB.

Syntax

```
CharSetForm getCharSetForm() const;
```

13.9.7 getCharSetId()

Returns the character set ID of the CLOB, in string form.

Syntax

```
string getCharSetId() const;
```

13.9.8 getCharSetIdUString()

Retrieves the characterset name associated with the Clob; UString version.

Syntax

```
UString getCharSetIdUString() const;
```

13.9.9 getChunkSize()

Returns the smallest data size to perform efficient writes to the CLOB.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
unsigned int getChunkSize() const;
```

13.9.10 getContentType()

Returns the content type of the `Clob`. If a content type has not been assigned, returns a `NULL` string.

Syntax

```
string getContentType();
```

13.9.11 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

```
LobOptionValue getOptions(
   LobOptionType optType);
```

Parameter	Description
`optType`	The `LobOptionType` setting requested. These may be combined using bitwise `or` (`|`) to avoid server round trips. See Table 7-1 and Table 7-2

13.9.12 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

```
Stream* getStream(
   unsigned int offset = 1,
   unsigned int amount = 0);
```

Parameter	Description
`offset`	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`.
`amount`	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`.

13.9.13 isInitialized()

Tests whether the `Clob object` is initialized. If the `Clob` object is initialized, `TRUE` is returned; otherwise, `FALSE` is returned.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool isInitialized() const;
```

13.9.14 isNull()

Tests whether the `Clob` object is atomically `NULL`. If the `Clob` object is atomically `NULL`, `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

13.9.15 isOpen()

Tests whether the `CLOB` is open. If the `CLOB` is open, `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isOpen() const;
```

13.9.16 length()

Returns the number of characters in the `CLOB`.

Syntax

```
unsigned int length() const;
```

13.9.17 open()

Opens the `CLOB` in `read/write` or read-only mode.

Syntax

```
void open(
   LObOpenMode mode = OCCI_LOB_READWRITE);
```

Parameter	Description
`mode`	`The mode the CLOB is to be opened in. Valid values are:` • `OCCI_LOB_READWRITE` • `OCCI_LOB_READONLY`

13.9.18 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

```
Clob& operator=(
   const Clob &srcClob);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
srcClob	The Clob from which the data must be copied.

13.9.19 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

```
bool operator==(
   const Clob &srcClob) const;
```

Parameter	Description
srcClob	The Clob object to be compared with the current Clob object.

13.9.20 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

```
bool operator!=(
   const Clob &srcClob) const;
```

Parameter	Description
srcClob	The Clob object to be compared with the current Clob object.

13.9.21 read()

Reads a part or all of the CLOB into a buffer.

Returns the actual number of characters read for fixed-width charactersets, such as UTF16, or the number of bytes read for multibyte charactersets, 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 charactersets (UTF16), or the number of bytes read for multibyte charactersets (including UTF8).

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`unsigned int read(`
`  unsigned int amt,`
`  unsigned char *buffer,`
`  unsigned int bufsize,`
`  unsigned int offset=1) const;`
	Reads a part or all of the CLOB into a buffer.

`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.

Parameter	Description
`amt`	The number of bytes to be read. from the CLOB.
`buffer`	The buffer that the CLOB data is to be read into.
`buffsize`	The size of the buffer. Valid values are numbers greater than or equal to `amt`.
`offset`	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.

13.9.22 setCharSetId()

Sets the Character set Id associated with `Clob`. The characterset 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

```
void setCharSetId(
   const string &charset);
```

Parameter	Description
`charset`	Oracle supported characterset name, such as `E8DEC`, `ZHT16BIG5`, or `OCCIUTF16`.

13.9.23 setCharSetIdUString()

Sets the characterset id associated with `Clob`; used when the environment's characterset is `UTF16`. The `charset` id set is used for read, write and getStream() operations.

Syntax

```
void setCharSetIdUSString(
   const string &charset);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
charset	Oracle supported characterset name, such as WE8DEC, ZHT16BIG5, or OCCIUTF16 in UString (UTF16 characterset).

13.9.24 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

```
void setCharSetForm(
   CharSetForm csfrm );
```

Parameter	Description
csfrm	The charset form for Clob.

13.9.25 setContentType()

Sets the content type of the Clob. If the Clob is not a SecureFile, throws an exception.

Syntax

```
void setContentType(
   const string contenttype);
```

Parameter	Description
contenttype	The content type of the Clob; an ASCII Mime compliant string.

13.9.26 setEmpty()

Sets the Clob object to empty.

Syntax	Description
void setEmpty();	Sets the Clob object to empty.
void setEmpty(const Connection* connectionp);	Sets the Clob object to empty and initializes the connection pointer to the passed parameter.

Parameter	Description
connectionp	The new connection pointer for the Clob object.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.9.27 setNull()

Sets the `Clob` object to atomically `NULL`.

Syntax

```
void setNull();
```

13.9.28 setOptions()

Specifies a `LobOptionValue` for a particular `LobOptionType`. Enables advanced compression, encryption and deduplication of CLOBs. See Table 7-1 and Table 7-2.

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

```
void setOptions(
   LobOptionType optType,
   LobOptionValue optValue);
```

Parameter	Description
`optType`	The `LobOptionType` setting being specified. These may be combined using bitwise `or` (`|`) to avoid server round trips.
`optValue`	The `LobOptionValue` setting for the `LobOptionType` specified by the `optType` parameter

13.9.29 trim()

Truncates the `CLOB to the new length specified.`

Syntax

```
void trim(
   unsigned int newlen);
```

Parameter	Description
`newlen`	The new length of the `CLOB`. Valid values are numbers less than or equal to the current length of the `CLOB`.

13.9.30 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`unsigned int write( unsigned int amt, unsigned char *buffer, unsigned int bufsize, unsigned int offset=1);`	Writes data from a buffer into a CLOB.
`unsigned int write( unsigned int amt, utext *buffer, unsigned int bufsize, unsigned int offset=1);`	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
amt	The amount parameter represents: • number of characters written for fixed-width charactersets (UTF16) • number of bytes written for multibyte charactersets (including UTF8)
buffer	The buffer containing the data to be written to the CLOB.
buffsize	The size of the buffer containing the data to be written to the CLOB. Valid values are numbers greater than or equal to amt.
offset	The starting position at which to begin writing data into 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.

13.9.31 writeChunk()

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.

Syntax	Description
`unsigned int writeChunk( unsigned int amt, unsigned char *buffer, unsigned int bufsize, unsigned int offset=1);`	Writes data from a buffer into a previously opened CLOB.
`unsigned int writeChunk( unsigned int amt, utext *buffer, unsigned int bufsize, unsigned int offset=1);`	Writes data from a UTF16 buffer into a CLOB; globalization enabled. Should be called after setting characterset to OCCIUTF16 using setCharSetIdUString() method.

Parameter	Description
amt	The amount parameter represents either a number of characters written for fixed-width charactersets (UTF16) or a number of bytes written for multibyte charactersets (including UTF8)

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
buffer	The buffer containing the data to be written to the CLOB.
buffsize	The size of the buffer containing the data to be written to the CLOB. Valid values are numbers greater than or equal to amt.
offset	The starting position at which to begin writing data into 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.

13.10 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

Attribute	Options
FailOverEventType	• FO_BEGIN indicates that a lost connection has been detected; failover is starting. • FO_END indicates that a failover completed successfully; the Connection is ready for use. • FO_ABORT indicates that the failover was unsuccessful; it is not be attempted again. • FO_REAUTH indicates that the user session has been reauthenticated. • FO_ERROR indicates that a failover was unsuccessful; the application can handle the error and retry failover.
FailOverType	• FO_NONE indicates that the user requested no protection for failover. • FO_SESSION indicates that the user requested only session failover. • FO_SELECT indicates that the use requested select failover.
ProxyType	• PROXY_DEFAULT is the database user name.

Table 13-12 Summary of Connection Methods

Method	Summary
changePassword()	Changes the password for the current user.
commit()	Commits changes made since the previous commit or rollback and release any database locks held by the session.
createStatement()	Creates a Statement object to execute SQL statements.
flushCache()	Flushes the object cache associated with the connection.
getClientCharSet()	Returns the default client character set.
getClientCharSetUString()	Returns the globalization enabled client character set in UString.
getClientNCHARCharSet()	Returns the default client NCHAR character set.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-12 (Cont.) Summary of Connection Methods

Method	Summary
getClientNCHARCharSetUString()	Returns the globalization enabled client NCHAR character set in UString.
getClientVersion()	Returns the version of the client used.
getLTXID()	Returns logical transaction id that may be used in various calls of package DBMS_APP_CONT.
getMetaData()	Returns the metadata for an object accessible from the connection.
getOCIServer()	Returns the OCI server context associated with the connection.
getOCIServiceContext()	Returns the OCI service context associated with the connection.
getOCISession()	Returns the OCI session context associated with the connection.
getServerVersion()	Returns the version of the Oracle server used, as string.
getServerVersionUString()	Returns the version of the Oracle server used, as a UString.
getStmtCacheSize()	Retrieves the size of the statement cache.
getTag()	Returns the tag associated with the connection.
isCached()	Determines if the specified statement is cached.
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.
postToSubscriptions()	Posts notifications to subscriptions.
readVectorOfBfiles()	Reads multiple Bfiles in a single server round-trip.
readVectorOfBlobs()	Reads multiple Blobs in a single server round-trip.
readVectorOfClobs()	Reads multiple Clobs in a single server round-trip.
registerSubscriptions()	Registers several Subscriptions for notification.
rollback()	Rolls back all changes made since the previous commit or rollback and release any database locks held by the session.
setStmtCacheSize()	Enables or disables statement caching.
setTAFNotify()	Registers failover callback function on the Connection object.
terminateStatement()	Closes a Statement object and free all resources associated with it.
unregisterSubscription()	Unregisters a Subscription, turns off its notifications.
writeVectorOfBlobs()	Writes multiple Blobs in a single server round-trip.
writeVectorOfClobs()	Writes multiple Clobs in a single server round-trip.

13.10.1 changePassword()

Changes the password of the user currently connected to the database.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void changePassword(`
`  const string &user,`
`  const string &oldPassword,`
`  const string &newPassword)=0;`
	Changes the password of the user.

`void changePassword(`
`  const UString &user,`
`  const UString &oldPassword,`
`  const UString &newPassword)=0;`
	Changes the password of the user (Unicode support). The client `Environment` should be initialized in OCCIUTIF16 mode.

Parameter	Description
`user`	The user currently connected to the database.
`oldPassword`	The current password of the user.
`newPassword`	The new password of the user.

13.10.2 commit()

Commits all changes made since the previous commit or rollback, and releases any database locks currently held by the session.

Syntax

```
void commit()=0;
```

13.10.3 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	Description
`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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`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
`sql`	The SQL string to be associated with the statement object.
`tag`	The tag whose associated statement must be retrieved from the cache. Ignored if statement caching is disabled.

13.10.4 flushCache()

Flushes the object cache associated with the connection.

Syntax

```
void flushCache()=0;
```

13.10.5 getClientCharSet()

Returns the session's character set.

Syntax

```
string getClientCharSet() const=0;
```

13.10.6 getClientCharSetUString()

Returns the globalization enabled client character set in `UString`.

Syntax

```
UString getClientCharSetUString() const=0;
```

13.10.7 getClientNCHARCharSet()

Returns the session's `NCHAR` character set.

Syntax

```
string getClientNCHARCharSet() const=0;
```

13.10.8 getClientNCHARCharSetUString()

Returns the globalization enabled client `NCHAR` character set in `UString`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
UString getClientNCHARCharSetUString() const=0;
```

13.10.9 getClientVersion()

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

```
void getClientVersion(
   int &majorVersion,
   int &minorVersion,
   int &updateNum,
   int &patchNumber,
   int &portUpdateNum)
```

Parameter	Description
`majorVersion`	The major version of the client library.
`minorVersion`	The minor version of the client library.
`updateNum`	The update number.
`patchNumber`	The number of the patch applied to the library.
`portUpdateNumber`	The number of the port-specific port update applied to the library.

13.10.10 getLTXID()

Returns logical transaction id that may be used in various calls of package `DBMS_APP_CONT`.

Syntax

```
Bytes getLTXID() const = 0
```

13.10.11 getMetaData()

Returns metadata for an object in the database.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`MetaData getMetaData(`
`  const string &object,`
`  MetaData::ParamType prmtyp=MetaData::PTYPE_UNK) const=0;`
	Returns metadata for an object in the database.

`MetaData getMetaData(`
`  const UString &object,`
`  MetaData::ParamType prmtyp=MetaData::PTYPE_UNK) const=0;`
	Returns metadata for a globalization enabled object in the database.

`MetaData getMetaData(` `  const RefAny &ref) const=0;`	Returns metadata for an object in the database through a reference.

Parameter	Description
object	The SQL string to be associated with the statement object.
prmtyp	The type of the schema object being described, as defined by the enumerated `ParamType` of the MetaData class, Table 13-27
ref	A REF to the Type Descriptor Object (TDO) of the type to be described.

13.10.12 getOCIServer()

Returns the OCI server context associated with the connection.

Syntax

```
OCIServer* getOCIServer() const=0;
```

13.10.13 getOCIServiceContext()

Returns the OCI service context associated with the connection.

Syntax

```
OCISvcCtx* getOCIServiceContext() const=0;
```

13.10.14 getOCISession()

Returns the OCI session context associated with the connection.

Syntax

```
OCISession* getOCISession() const=0;
```

13.10.15 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.10.16 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;
```

13.10.17 getStmtCacheSize()

Retrieves the size of the statement cache.

Syntax

```
unsigned int getStmtCacheSize() const=0;
```

13.10.18 getTag()

Returns the tag associated with the connection. Valid only for connections from a stateless connection pool.

Syntax

```
string getTag() const=0;
```

13.10.19 isCached()

Determines if the specified statement is cached.

Syntax	Description

`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
`sql`	The SQL string to be associated with the statement object.
`tag`	The tag whose associated statement must be retrieved from the cache. Ignored if statement caching is disabled.

13.10.20 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`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
`conn`	Connection
`vect`	Vector of `Ref` objects that are pinned.
`vectObj`	Vector that contains objects after the pinning operation is complete; an `OUT` parameter.
`lockOpt`	Lock option used during the pinning of the array, as defined by `LockOptions` in Table 13-2. The only supported value is `OCCI_LOCK_NONE`.

13.10.21 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 Database Advanced Queuing functionality.

Syntax

```
void postToSubscriptions(
   const vector<aq::Subscription>& sub)=0;
```

Parameter	Description
`sub`	The vector of subscriptions that receive postings.

13.10.22 readVectorOfBfiles()

Reads multiple `Bfiles` in a single server round-trip. All `Bfiles` must be open for reading.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void readVectorOfBfiles(
   const Connection *conn,
   vector<Bfile> &vec,
   oraub8 *byteAmts,
   oraub8 *offsets,
   unsigned char *buffers[],
   oraub8 *bufferLengths);
```

Parameter	Description
`conn`	Connection.
`vec`	Vector of `Bfile` objects; each `Bfile` must be open for reading.
`byteAmts`	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.
`offsets`	Array of offsets, starting position where reading from the `Bfiles` starts.
`buffers`	Array of pointers to buffers into which the data is read.
`bufferLengths`	Array of sizes of each buffer, in bytes.

13.10.23 readVectorOfBlobs()

Reads multiple `BLOBs` in a single server round-trip.

Syntax

```
void readVectorOfBlobs(
   const Connection *conn,
   vector<Blob> &vec,
   oraub8 *byteAmts,
   oraub8 *offsets,
   unsigned char *buffers[],
   oraub8 *bufferLengths);
```

Parameter	Description
`conn`	Connection.
`vec`	Vector of `Blob` objects.
`byteAmts`	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.
`offsets`	Array of offsets, starting position where reading from the `Blobs` starts.
`buffers`	Array of pointers to buffers into which the data is read.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
bufferLengths	Array of sizes of each buffer, in bytes.

13.10.24 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	Description

`void readVectorOfClobs(`
`  const Connection *conn,`
`  vector<Clob> &vec,`
`  oraub8 *byteAmts,`
`  araub8 *charAmts,`
`  oraub8 *offsets,`
`  unsigned char *buffers[],`
`  oraub8 *bufferLengths);`
	General form of the method.

`void readVectorOfClobs(`
`  const Connection *conn,`
`  vector<Clob> &vec,`
`  oraub8 *byteAmts,`
`  araub8 *charAmts,`
`  oraub8 *offsets,`
`  utext *buffers[],`
`  oraub8 *bufferLengths);`
	Form of the method used with utext buffers, when data is in UTF16 characterset encoding.

Parameter	Description
conn	Connection.
vec	Vector of Clob objects.
byteAmts	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.
charAmts	Array of amount of characters to read from individual Clobs. Returns the number of characters read for each Clob.
offsets	Array of offsets, starting position where reading from the Clobs starts, in characters.
buffers	Array of pointers to buffers into which the data is read.
bufferLengths	Array of sizes of each buffer, in bytes.

13.10.25 registerSubscriptions()

Registers Subscriptions for notification.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

```
void registerSubscriptions(
   const vector<aq::Subscription>& sub)=0;
```

Parameter	Description
`sub`	Vector of subscriptions that are registered for notification.

13.10.26 rollback()

Drops all changes made since the previous commit or rollback, and releases any database locks currently held by the session.

Syntax

```
void rollback()=0;
```

13.10.27 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;
```

Parameter	Description
`cacheSize`	The maximum number of statements in the cache.

13.10.28 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. 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

```
void setTAFNotify(
   int (*notifyFn)(
      Environment *env,
      Connection *conn,
      void *ctx,
      FailOverType foType,
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
    FailOverEventType foEvent),
  void *ctxTAF)
```

Parameter	Description
notifyFn	The user defined callback function invoked during failover events.
env	Environment object from which the failing Connection was created.
conn	The failing Connection on which the callback function is registered.
ctx	Context supplied by the user when registering the callback.
foType	The configured FailOverType, values FO_SESSION or FO_SELECT, as defined in Table 13-11.
foEvent	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.
ctxTAF	User context passed back to the callback function at invocation.

13.10.29 terminateStatement()

Closes a Statement object.

Syntax	Description
`void terminateStatement(` `  Statement *stmt)=0;`	Closes a Statement object and frees all resources associated with it.

`void terminateStatement(`
`  Statement *stmt,`
`  const string &tag)=0;`
	Releases statement back to the cache after adding an optional tag, a string.

`void terminateStatement(`
`  Statement* stmt,`
`  const UString &tag) = 0;`
	Releases statement back to the cache after adding an optional tag, a UString.

Parameter	Description
stmt	The Statement to be closed.
tag	The tag associated with the statement, either a string or a UString.

13.10.30 unregisterSubscription()

Unregisters a Subscription, turning off its notifications.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void unregisterSubscription(
   const aq::Subscription& sub)=0;
```

Parameter	Description
sub	Subscription whose notifications is turned off.

13.10.31 writeVectorOfBlobs()

Writes multiple Blobs in a single server round-trip.

Syntax

```
void writeVectorOfBlobs(
   const Connection *conn,
   vector<Blob> &vec,
   oraub8 *byteAmts,
   oraub8 *offsets,
   unsigned char *buffers[],
   oraub8 *bufferLengths);
```

Parameter	Description
conn	Connection.
vec	Vector of Blob objects.
byteAmts	Array of amount of bytes to write to the individual Blobs.
offsets	Array of offsets, starting position where writing to the Blobs starts.
buffers	Array of pointers to buffers from which the data is written.
bufferLengths	Array of sizes of each buffer, in bytes.

13.10.32 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void writeVectorOfClobs(`
`  const Connection *conn,`
`  vector<Clob> &vec,`
`  oraub8 *byteAmts,`
`  araub8 *charAmts,`
`  oraub8 *offsets,`
`  unsigned char *buffers[],`
`  oraub8 *bufferLengths);`
	General form of the method.

`void writeVectorOfClobs(`
`  const Connection *conn,`
`  vector<Clob> &vec,`
`  oraub8 *byteAmts,`
`  araub8 *charAmts,`
`  oraub8 *offsets,`
`  utext *buffers[],`
`  oraub8 *bufferLengths);`
	Form of the method used with `utext` buffers, when data is in UTF16 characterset encoding.

Parameter	Description
`conn`	Connection.
`vec`	Vector of `Clob` objects.
`byteAmts`	Array of amount of bytes to write to the individual `Clob`s. Only used if the `charAmts` is `NULL` or `0` for any `Clob` index. Returns the number of bytes written for each `Clob`.
`charAmts`	Array of amount of characters to write to individual `Clob`s. Returns the number of characters read for each `Clob`.
`offsets`	Array of offsets, starting position where writing to the `Clob`s starts, in characters.
`buffers`	Array of pointers to buffers from which the data is written.
`bufferLengths`	Array of sizes of each buffer, in bytes.

13.11 ConnectionPool Class

The ConnectionPool class represents a pool of connections for a specific database.

Table 13-13 Summary of ConnectionPool Methods

Method	Summary
createConnection()	Creates a pooled connection.
createProxyConnection()	Creates a proxy connection.
getBusyConnections()	Returns the number of busy connections in the connection pool.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-13 (Cont.) Summary of ConnectionPool Methods

Method	Summary
getIncrConnections()	Returns the number of incremental connections in the connection pool.
getMaxConnections()	Returns the maximum number of connections in the connection pool.
getMinConnections()	Returns the minimum number of connections in the connection pool.
getOpenConnections()	Returns the number of open connections in the connection pool.
getPoolName()	Returns the name of the connection pool.
getStmtCacheSize()	Retrieves the size of the statement cache.
getTimeOut()	Returns the time out period for a connection in the connection pool.
setErrorOnBusy()	Specifies that a `SQLException` should be generated when all connections in the connection pool are busy and no further connections can be opened.
setPoolSize()	Sets the minimum, maximum, and incremental number of pooled connections for the connection pool.
setStmtCacheSize()	Enables or disables statement caching.
setTimeOut()	Sets the time out period, in seconds, for a connection in the connection pool.
terminateConnection()	Destroys the connection.

13.11.1 createConnection()

Creates a pooled connection.

Syntax	Description

`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.

`Connection* createConnection(`
`  const UString &username,`
`  const UString &password)=0;`
	Creates a globalization enabled pooled connection.

Parameter	Description
`userName`	The name of the user with which to connect.
`password`	The password of the user.

13.11.2 createProxyConnection()

Creates a proxy connection from the connection pool.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`Connection* createProxyConnection( const string &name, Connection::ProxyType proxyType=Connection::PROXY_DEFAULT)=0;`	Creates a proxy connection.
`Connection* createProxyConnection( const UString &name, Connection::ProxyType proxyType=Connection::PROXY_DEFAULT)=0;`	Creates a globalization enabled proxy connection.
`Connection* createProxyConnection( const string &name, string roles[], int numRoles, Connection::ProxyType proxyType=Connection::PROXY_DEFAULT)=0;`	Creates a proxy connection for several roles.
`Connection* createProxyConnection( const UString &name, string roles[], unsigned int numRoles, Connection::ProxyType proxyType=Connection::PROXY_DEFAULT)=0;`	Creates a globalization enabled proxy connection for several roles.

Parameter	Description
`name`	The user name to connect with.
`roles`	The roles to activate on the database server.
`numRoles`	The number of roles to activate on the database server.
`proxyType`	The type of proxy authentication to perform, `ProxyType`, defined in Table 13-11. Valid values are: • `PROXY_DEFAULT` representing a database user name.

13.11.3 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

```
unsigned int getBusyConnections() const=0;
```

13.11.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
unsigned int getIncrConnections() const=0;
```

13.11.5 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

```
unsigned int getMaxConnections() const=0;
```

13.11.6 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

```
unsigned int getMinConnections() const=0;
```

13.11.7 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

```
unsigned int getOpenConnections() const=0;
```

13.11.8 getPoolName()

Returns the name of the connection pool.

Syntax

```
string getPoolName() const=0;
```

13.11.9 getStmtCacheSize()

Retrieves the size of the statement cache.

Syntax

```
unsigned int getStmtCacheSize() const=0;
```

13.11.10 getTimeOut()

Returns the time out period of a connection in the connection pool.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
unsigned int getTimeOut() const=0;
```

13.11.11 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;
```

13.11.12 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;
```

Parameter	Description
`minConn`	The minimum number of connections for the connection pool.
`maxConn`	The maximum number of connections for the connection pool.
`incrConn`	The incremental number of connections for the connection pool.

13.11.13 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;
```

Parameter	Description
`cacheSize`	The size of the statement cache.

13.11.14 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

```
void setTimeOut(
   unsigned int connTimeOut = 0)=0;
```

Parameter	Description
`connTimeOut`	The timeout period in number of seconds.

13.11.15 terminateConnection()

Terminates the pooled connection or proxy connection.

Syntax

```
void terminateConnection(
   Connection *connection)=0;
```

Parameter	Description
`connection`	The pooled connection or proxy connection to terminate.

13.12 Consumer Class

The `Consumer` class supports dequeuing of `Messages` and controls the dequeuing options.

Table 13-14 Enumerated Values Used by Consumer Class

Attribute	Options
`DequeMode`	• `DEQ_BROWSE` indicates that the message should be read without acquiring a lock; equivalent to a `SELECT`. • `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`. • `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. • `DEQ_REMOVE_NODATA` indicates that the receipt of the message should be confirmed, but its actual content should not be delivered.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-14 (Cont.) Enumerated Values Used by Consumer Class

Attribute	Options
`Navigation`	• `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. • `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. • `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.
`Visibility`	• `DEQ_IMMEDIATE` indicates that the dequeued message is not part of the current transaction. It constitutes a transaction on its own. • `DEQ_ON_COMMIT` indicates that the dequeue is part of the current transaction. This is the default setting.
`DequeWaitOption`	• `DEQ_WAIT_FOREVER` indicates that the consumer waits for the `Message` indefinitely. • `DEQ_NO_WAIT` indicates that there should be not wait if there are no messages on the queue.

Table 13-15 Summary of Consumer Methods

Method	Description
Consumer()	`Consumer` class constructor.
getConsumerName()	Retrieves the name of the `Consumer`.
getCorrelationId()	Retrieves she correlation id of the message that is to be dequeued.
getDequeueMode()	Retrieves the dequeue mode of the `Consumer`.
getMessageIdToDequeue()	Retrieves the id of the message that is dequeued.
getQueueName()	Gets the name of the queue used by the consumer.
getPositionOfMessage()	Retrieves the position of the Message that is dequeued.
getTransformation()	Retrieves the transformation applied before a `Message` is dequeued.
getVisibility()	Retrieves the transactional behavior of the dequeue operation.
getWaitTime()	Retrieves the specified behavior of the `Consumer` when waiting for a `Message` with matching search criteria.
isNull()	Tests whether the `Consumer` object is `NULL`.
operator=()	Assignment operator for the `Consumer` class.
receive()	Receives and dequeues a `Message`
setAgent()	Sets the `Agent`'s name and address (queue name) on the consumer.
setConsumerName()	Sets the `Consumer` name.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-15 (Cont.) Summary of Consumer Methods

Method	Description
setCorrelationId()	Specifies the correlation identifier of the message to be dequeued.
setDequeueMode()	Specifies the locking behavior associated with dequeuing.
setMessageIdToDequeue()	Specifies the identifier of the `Message` to be dequeued.
setNull()	Nullifies the `Consumer`; frees the memory associated with this object.
setPositionOfMessage()	Specifies position of the `Message` to be retrieved.
setQueueName()	Specifies the name of a queue before dequeuing `Messages`.
setTransformation()	Specifies transformation applied before dequeuing a `Message`.
setVisibility()	Specifies if `Message` should be dequeued as part of the current transaction.
setWaitTime()	Specifies wait conditions if there are no `Messages` with matching criteria.

13.12.1 Consumer()

Consumer class constructor.

Syntax	Description
`Consumer(` `  const Connection *conn);`	Creates a new `Consumer` object with the specified `Connection` handle.

`Consumer(`
`  const Connection *conn`
`  const Agent& agent);`
	Creates a new `Consumer` object with specified `Connection` and properties of the specified `Agent`.

`Consumer(`
`  const Connection *conn,`
`  const string& queue);`
	Creates a new `Consumer` object with specified `Connection` and queue.

`Consumer(` `  const Consumer& consumer);`	Copy constructor.

Parameter	Description
`conn`	The connection in which the `Consumer` is created.
`agent`	Agent assigned to the `Consumer`.
`queue`	Queue at which the `Consumer` retrieves messages.
`consumer`	Original `Consumer` object.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.12.2 getConsumerName()

Retrieves the name of the `Consumer`.

Syntax

```
string getConsumerName() const;
```

13.12.3 getCorrelationId()

Retrieves she correlation id of the message that is to be dequeued

Syntax

```
string geCorrelationId() const;
```

13.12.4 getDequeueMode()

Retrieves the dequeue mode of the `Consumer`. `DequeueMode` is defined in Table 13-14.

Syntax

```
DequeueMode getDequeueMode() const;
```

13.12.5 getMessageIdToDequeue()

Retrieves the id of the message that is dequeued.

Syntax

```
Bytes getMessageToDequeue() const;
```

13.12.6 getPositionOfMessage()

Retrieves the position, or navigation, of the message that is dequeued. `Navigation` is defined in Table 13-14.

Syntax

```
Navigation getPositionOfMessage() const;
```

13.12.7 getQueueName()

Gets the name of the queue used by the consumer.

Syntax

```
string getQueueName() const;
```

13.12.8 getTransformation()

Retrieves the transformation applied before a `Message` is dequeued.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
string getTransformation() const;
```

13.12.9 getVisibility()

Retrieves the transactional behavior of the dequeue operation, or visibility. `Visibility` is defined in Table 13-14.

Syntax

```
Visibility getVisibility() const;
```

13.12.10 getWaitTime()

Retrieves the specified behavior of the `Consumer` when waiting for a `Message` with matching search criteria. `DequeWaitOption` is defined in Table 13-14.

Syntax

```
DequeWaitOption getWaitTime() const;
```

13.12.11 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;
```

13.12.12 operator=()

Assignment operator for `Consumer` class.

Syntax

```
void operator=(
   const Consumer& consumer);
```

Parameter	Description
`consumer`	The original `Consumer`.

13.12.13 receive()

Receives and dequeues a `Message`.

Syntax

```
Message receive(
   Message::PayloadType pType,
   const string& type="",
   const string& schema="");
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
pType	The type of payload expected. Payload Type is defined in Table 13-14.
type	The type of the payload when type is OBJECT.
schema	The schema in which the type is defined when pType is OBJECT.

13.12.14 setAgent()

Sets the Agent's name and address (queue name) on the consumer.

Syntax

```
void setAgent(
   const Agent& agent);
```

Parameter	Description
agent	Name of the Agent.

13.12.15 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

```
void setConsumerName(
   const string& name);
```

Parameter	Description
name	Name of the Consumer.

13.12.16 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

```
void setCorrelationId
   const string& id);
```

Parameter	Description
id	The identifier of the Message.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.12.17 setDequeueMode()

Specifies the locking behavior associated with dequeuing.

Syntax

```
void setDequeueMode(
   DequeueMode mode);
```

Parameter	Description
mode	Behavior of enqueuing. DequeMode is defined in Table 13-14.

13.12.18 setMessageIdToDequeue()

Specifies the identifier of the Message to be dequeued.

Syntax

```
void setMessageIdToDequeue(
   const Bytes& msgid);
```

Parameter	Description
msgid	Identifier of the Message to be dequeued.

13.12.19 setNull()

Nullifies the Consumer; frees the memory associated with this object.

Syntax

```
void setNull();
```

13.12.20 setPositionOfMessage()

Specifies position of the Message to be retrieved.

Syntax

```
void setPositionOfMessage(
   Navigation pos);
```

Parameter	Description
pos	Position of the message, Navigation, is defined in Table 13-14.

13.12.21 setQueueName()

Specifies the name of a queue before dequeuing Messages. Typically used when dequeuing multiple messages from the same queue.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setQueueName(
   const string& queue);
```

Parameter	Description
queue	The name of a valid queue in the database.

13.12.22 setTransformation()

Specifies transformation applied before dequeuing the Message.

Syntax

```
void setTransformation(
   string &fName);
```

Parameter	Description
fName	SQL transformation function.

13.12.23 setVisibility()

Specifies if Message should be dequeued as part of the current transaction. Visibility parameter is ignored when in DEQ_BROWSE mode.

Syntax

```
void setVisibility(
   Visibility option);
```

Parameter	Description
option	Visibility option being set, defined in Table 13-14.

13.12.24 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

```
void setWaitTime(
   DequeWaitOption wait);
```

Parameter	Description
wait	Waiting conditions. DequeWaitOption is defined in Table 13-14.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.13 Date Class

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.

```
/* 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

Method	Summary
Date()	`Date` class constructor.
addDays()	Returns a `Date` object with *n* days added.
addMonths()	Returns a `Date` object with *n* months added.
daysBetween()	Returns the number of days between the current `Date` object and the date specified.
fromBytes()	Convert an external `Bytes` representation of a `Date` object to a `Date` object.
fromText()	Convert the date from a given input string with format and NLS parameters specified.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-16 (Cont.) Summary of Date Methods

Method	Summary
getDate()()	Returns the date and time components of the `Date` object.
getSystemDate()	Returns a `Date` object containing the system date.
isNull()	Returns `TRUE` if `Date` is `NULL; otherwise returns false`.
lastDay()	Returns a `Date` that is the last day of the month.
nextDay()	Returns a `Date` that is the date of the next day of the week.
operator=()	Assigns the values of a date to another.
operator==()	Returns `TRUE` if `a` and `b` are the same, `false` otherwise.
operator!=()	Returns `TRUE` if `a` and `b` are unequal, `false` otherwise.
operator>()	Returns `TRUE` if `a` is past `b`, `false` otherwise.
operator>=()	Returns `TRUE` if `a` is past `b or equal to b`, `false` otherwise.
operator=()	Returns `TRUE` if `a` is before `b`, `false` otherwise.
operator>()	Returns `TRUE` if `a` is before `b`, or equal to `b`, `false` otherwise.
setDate()	Sets the date from the date components input.
setNull()	Sets the object state to `NULL`.
toBytes()	Converts the `Date` object into an external `Bytes` representation.
toText()	Returns the `Date` object as a string.
toZone()	Returns a `Date` object converted from one time zone to another.

13.13.1 Date()

`Date` class constructor.

Syntax	Description
`Date();`	Creates a `NULL Date` object.
`Date(` `  const Date &srcDate);`	Creates a copy of a `Date` object.

`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);`
	Creates a `Date` object using integer parameters.

Parameter	Description
year	-4712 to 9999, except 0
month	1 to 12

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
day	1 to 31
minutes	0 to 59
seconds	0 to 59

13.13.2 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.

13.13.3 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.

13.13.4 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.

13.13.5 fromBytes()

Converts a `Bytes` object to a `Date` object.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void fromBytes(
   const Bytes &byteStream,
   const Environment *envp = NULL);
```

Parameter	Description
`byteStream`	`Date` in external format in the form of Bytes.
`envp`	The OCCI environment.

13.13.6 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

`void fromText(`
`  const string &datestr,`
`  const string &fmt = "",`
`  const string &nlsParam = "",`
`  const Environment *envp = NULL);`
	Sets `Date` object to value represented by a `string`.

`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
`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. 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.13.7 getDate()

Returns the date in the form of the date components year, month, day, hour, minute, seconds.

Syntax

```
void getDate(
   int &year,
   unsigned int &month,
   unsigned int &day,
   unsigned int &hour,
   unsigned int &min,
   unsigned int &seconds) const;
```

Parameter	Description
`year`	The year component of the date.
`month`	The month component of the date.
`day`	The day component of the date.
`hour`	The hour component of the date.
`min`	The minutes component of the date.
`seconds`	The seconds component of the date.

13.13.8 getSystemDate()

Returns the system date.

Syntax

```
static Date getSystemDate(
   const Environment *envp);
```

Parameter	Description
`envp`	The environment in which the system date is returned.

13.13.9 isNull()

Tests whether the `Date` is `NULL`. If the `Date` is `NULL`, `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.13.10 lastDay()

Returns a date representing the last day of the current month.

Syntax

```
Date lastDay() const;
```

13.13.11 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`

Syntax	Description
`Date nextDay( const string &dow) const;`	Returns a date representing the day after the day of the week specified.
`Date nextDay( const UString &dow) const;`	Returns a date representing the day after the day of the week specified.; globalization enabled. The parameter should be in the character set associated with the environment from which the date was created.

Parameter	Description
`dow`	A string representing the day of the week.

13.13.12 operator=()

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

```
Date& operator=(
   const Date &d);
```

Parameter	Description
`date`	The date object that is assigned.

13.13.13 operator==()

Compares the dates specified. If the dates are equal, `TRUE` is returned; otherwise, `FALSE` is returned.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool operator==(
   const Date &first,
   const Date &second);
```

Parameter	Description
`first`	The first date to be compared.
`second`	The second date to be compared.

13.13.14 operator!=()

Compares the dates specified. If the dates are not equal then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool operator!=(
   const Date &first,
   const Date &second);
```

Parameter	Description
`first`	The first date to be compared.
`second`	The second date to be compared.

13.13.15 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

```
bool operator>(
   const Date &first,
   const Date &second);
```

Parameter	Description
`first`	The first date to be compared.
`second`	The second date to be compared.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.13.16 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

```
bool operator>=(
   const Date &first,
   const Date &second);
```

Parameter	Description
`first`	The first date to be compared.
`second`	The second date to be compared.

13.13.17 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

```
bool operator<(
   const Date &first,
   const Date &second);
```

Parameter	Description
`first`	The first date to be compared.
`second`	The second date to be compared.

13.13.18 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

```
bool operator<=(
   const Date &first,
   const Date &second);
```

Parameter	Description
`first`	The first date to be compared.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
second	The second date to be compared.

13.13.19 setDate()

Sets the date to the values specified.

Syntax

```
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
year	The argument specifying the year value. Valid values are -4713 through 9999.
month	The argument specifying the month value. Valid values are 1 through 12.
day	The argument specifying the day value. Valid values are 1 through 31.
hour	The argument specifying the hour value. Valid values are 0 through 23.
min	The argument specifying the minutes value. Valid values are 0 through 59.
seconds	The argument specifying the seconds value. Valid values are 0 through 59.

13.13.20 setNull()

Sets the Date to atomically NULL.

Syntax

```
void setNull();
```

13.13.21 toBytes()

Returns the date in Bytes representation.

Syntax

```
Bytes toBytes() const;
```

13.13.22 toText()

Returns a string or UString with the value of this date formatted using fmt and nlsParam.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

`string toText(`
`  const string &fmt = "",`
`  const string &nlsParam = "") const;`
	Returns a `string` with the value of this date formatted using `fmt` and `nlsParam`.

`UString toText(`
`  const UString &fmt,`
`  const UString &nlsParam) const;`
	Returns a `UString` with the value of this date formatted using `fmt` and `nlsParam`.

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`.

13.13.23 toZone()

Returns `Date` value converted from one time zone to another.

Syntax

```
Date toZone(
   const string &zone1,
   const string &zone2) const;
```

Parameter	Description
`zone1`	A string representing the time zone to be converted from.
`zone2`	A string representing the time zone to be converted to.

Valid time zone codes are:

Zone code	Value
`AST, ADT`	Atlantic Standard or Daylight Time
`BST, BDT`	Bering Standard or Daylight Time
`CST, CDT`	Central Standard or Daylight Time
`EST, EDT`	Eastern Standard or Daylight Time

Copyright © 1999, 2025, Oracle and/or its affiliates.

Zone code	Value
GMT	Greenwich Mean Time
HST, HDT	Alaska-Hawaii Standard Time or Daylight Time
MST, MDT	Mountain Standard or Daylight Time
NST	Newfoundland Standard Time
PST, PDT	Pacific Standard or Daylight Time
YST, YDT	Yukon Standard or Daylight Time

13.14 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

Attribute	Options
Mode	• DEFAULT is used for creating an Environment object; it has no thread safety or object support. • OBJECT is for creating an Environment object; it uses object features. • SHARED is for creating an Environment object. • NO_USERCALLBACKS is for creating an Environment object; it does not support user callbacks. • THREADED_MUTEXED is a thread safe mode for creating an Environment object, mutexed internally by OCCI. • THREADED_UNMUTEXED is a thread safe mode for creating an Environment object; the client is responsible for mutexing. • EVENTS supports registration for event notification used in Oracle Database Advanced Queuing. • USE_LDAP supports registration with LDAP.

Table 13-18 Summary of Environment Methods

Method	Summary
createConnection()	Establishes a connection to the specified database.
createConnectionPool()	Creates a connection pool.
createEnvironment()	Creates an Environment object.
createStatelessConnectionPool()	Creates a stateless connection pool.
enableSubscription()	Enables subscription notification
disableSubscription()	Disables subscription notification
getCacheMaxSize()	Retrieves the Cache Max heap size.
getCacheOptSize()	Retrieves the cache optimal heap size.
getCacheSortedFlush()	Retrieves the setting of the cache sorting flag.
getClientVersion()	Returns the version of the client library.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-18 (Cont.) Summary of Environment Methods

Method	Summary
getCurrentHeapSize()	Returns the current amount of memory allocated to all objects in the current environment.
getLDAPAdminContext()	Returns the administrative context when using LDAP open notification registration.
getLDAPAuthentication()	Returns the authentication mode when using LDAP open notification registration.
getLDAPHost()	Returns the host on which the LDAP server runs.
getLDAPPort()	Returns the port on which the LDAP server is listening.
getMap()()	Returns the Map for the current environment.
getNLSLanguage()	Returns the NLS Language for the current environment.
getNLSTerritory()	Returns the NLS Territory for the current environment.
getOCIEnvironment()	Returns the OCI environment associated with the current environment.
getXAConnection()	Creates an XA connection to a database.
getXAEnvironment()	Creates an `XA Environment` object.
releaseXAConnection()	Releases all resources allocated by a getXAConnection() call.
releaseXAEnvironment()	Releases all resources allocated by a getXAEnvironment() call.
setCacheMaxSize()	Specifies the maximum size for the client-side object cache as a percentage of the optimal size.
setCacheOptSize()	Specifies the optimal size for the client-side object cache in bytes.
setCacheSortedFlush()	Specifies whether to sort cache in table order before flushing.
setLDAPAdminContext()	Specifies the administrative context for the LDAP client.
setLDAPAuthentication()	Specifies the LDAP authentication mode.
setLDAPHostAndPort()	Specifies the LDAP server host and port.
setLDAPLoginNameAndPassword()	Specifies the login name and password when connecting to an LDAP server.
setNLSLanguage()	Specifies the NLS Language for the current environment.
setNLSTerritory()	Specifies the NLS Territory for the current environment.
terminateConnection()	Closes the connection pool and free all related resources.
terminateConnectionPool()	Closes the connection pool and free all related resources.
terminateEnvironment()	Destroys the environment.
terminateStatelessConnectionPool()	Closes the stateless connection pool and free all related resources.

13.14.1 createConnection()

This method establishes a connection to the database specified.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`Connection * createConnection(`
`  const string &userName,`
`  const string &password,`
`  const string &connectString="")=0;`
	Creates a default connection; if the `userName` and `password` are `NULL`, the connection may be authenticated externally.

`Connection * createConnection(`
`  const UString &userName,`
`  const UString &password,`
`  const UString &connectString)=0;`
	Creates a connection (Unicode support). The client `Environment` should be initialized in `OCCIUTIF16` mode.

`Connection * createConnection(`
`  const string &userName,`
`  const string &password,`
`  const string &connectString,`
`  const string &connectionClass,`
`  const Connection::Purity &purity)=0;`
	Creates a connection for database resident connection pooling.

`Connection * createConnection(`
`  const UString &userName,`
`  const UString &password,`
`  const UString &connectString,`
`  const UString &connectionClass`
`  const Connection::Purity &purity)=0;`
	Creates a connection for database resident connection pooling (Unicode support). The client `Environment` should be initialized in `OCCIUTIF16` mode.

`Connection * createConnection(const OCCI_STD_NAMESPACE::string &connectString);`	`Here, ConnectString`, must be a Centralized Configuration Provider URL. The Centralized Configuration Provider contains configuration parameters and credentials which can be fetched by OCCI. The Centralized Configuration Provider data resides on either the Azure Cloud or Oracle Cloud Infrastructure.
`Connection * createConnection(const UString &connectString);`	This constructor supports URLs with UTF16 character encoded.
`Connection * createConnection();`	This constructor supports URLs in `TWO_TASK`, `TWO_TASK` contains the `connectString` URL to the Cloud Centralized Configuration Provider.

Parameter	Description
`userName`	The name of the user with which to connect.
`password`	The password of the user.
`connectString`	The database to which the connection is made.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
purity	The purity of the connection used for database resident connection pooling; either SELF or NEW.
connectionClass	The connection class used for database resident connection pooling.

> **See Also**
>
> Configuring the Centralized Configuration Provider Naming Method

13.14.2 createConnectionPool()

Creates a connection pool based on the parameters specified.

Syntax	Description

`ConnectionPool* createConnectionPool(`
`  const string &poolUserName,`
`  const string &poolPassword,`
`  const string &connectString = "",`
`  unsigned int minConn = 0,`
`  unsigned int maxConn = 1,`
`  unsigned int incrConn = 1)=0;`
	Creates a default connection pool.

`ConnectionPool* createConnectionPool(`
`  const UString &poolUserName,`
`  const UString &poolPassword,`
`  const UString &connectString,`
`  unsigned int minConn = 0,`
`  unsigned int maxConn = 1,`
`  unsigned int incrConn = 1)=0;`
	Creates a connection pool (Unicode support). The client Environment should be initialized in OCCIUTIF16 mode.

Parameter	Description
poolUserName	The pool user name.
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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
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.

13.14.3 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, then all three memory management functions must be thread-safe.

Syntax	Description
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);	Creates a default environment.
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);	Creates an environment with the specified character set and NCHAR character set ids (Unicode support). The client Environment should be initialized in OCCIUTIF16 mode.

Parameter	Description
mode	Values are defined as part of Mode in Table 13-17: DEFAULT, THREADED_MUTEXED, THREADED_UNMUTEXED, OBJECT.
ctxp	Context pointer for user-defined memory management function.
size	The size of the memory allocated by user-defined memory allocation function.
newsize	The new size of the memory to be reallocated.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
memptr	The existing memory that must be reallocated to new size.
malocfp	User-defined memory allocation function.
ralocfp	User-defined memory reallocation function.
mfreefp	User-defined memory free function.
charset	Character set id that replaces the one specified in NLS_LANG.
ncharset	Character set id that replaces the one specified in NLS_NCHAR.

13.14.4 createStatelessConnectionPool()

Creates a StatelessConnectionPool object with specified pool attributes.

Syntax	Description
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.
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
poolUserName	The pool user name.
poolPassword	The pool password.
connectString	The connection string for the server.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
maxConn	The maximum number of connections that can be opened the pool; additional sessions cannot be open.
minConn	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 .
incrConn	The number of connections by which to increment the pool if all open connections are busy, up to a maximum open connections specified by maxConn parameter. This parameter is considered only if the PoolType is set to HOMOGENEOUS, as defined in Table 13-41 .
pType	The PoolType of the connection pool, defined in Table 13-41 .

13.14.5 enableSubscription()

Enables subscription notification.

Syntax

```
void enableSubscription(
   const aq::Subscription &sub);
```

Parameter	Description
sub	The Subscription.

13.14.6 disableSubscription()

Disables subscription notification.

Syntax

```
void disableSubscription(
   Subscription &subscr);
```

Parameter	Description
subscr	The Subscription.

13.14.7 getCacheMaxSize()

Retrieves the maximum size of the cache.

Syntax

```
unsigned int getCacheMaxSize() const;
```

13.14.8 getCacheOptSize()

Retrieves the Cache optimal heap size.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
unsigned int getCacheOptSize() const;
```

13.14.9 getCacheSortedFlush()

Retrieves the current setting of the cache sorting flag; `TRUE` or `FALSE`.

Syntax

```
bool getCacheSortedFlush() const;
```

13.14.10 getCurrentHeapSize()

Returns the amount of memory currently allocated to all objects in this environment.

Syntax

```
unsigned int getCurrentHeapSize() const;
```

13.14.11 getLDAPAdminContext()

Returns the administrative context when using LDAP open notification registration.

Syntax

```
string getLDAPAdminContext() const;
```

13.14.12 getLDAPAuthentication()

Returns the authentication mode when using LDAP open notification registration.

Syntax

```
unsigned int getLDAPAuthentication() const;
```

13.14.13 getLDAPHost()

Returns the host on which the LDAP server runs.

Syntax

```
string getLDAPHost() const;
```

13.14.14 getLDAPPort()

Returns the port on which the LDAP server is listening.

Syntax

```
unsigned int getLDAPPort() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.14.15 getMap()

Returns a pointer to the map for this environment.

Syntax

```
Map *getMap() const;
```

13.14.16 getNLSLanguage()

Returns the NLS Language for the current environment.

Syntax

```
string getNLSLanguage() const;
```

13.14.17 getNLSTerritory()

Returns the NLS Territory for the current environment.

Syntax

```
string getNLSTerritory() const;
```

13.14.18 getOCIEnvironment()

Returns a pointer to the OCI environment associated with this environment.

Syntax

```
OCIEnv *getOCIEnvironment() const;
```

13.14.19 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);
```

Parameter	Description
`dbname`	The database name; same as the optional `dbname` provided in the Open String (and used in connection to the Resource Manager).

13.14.20 getXAEnvironment()

Returns a pointer to an OCCI Environment object that corresponds to the one opened by the XA library.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
Environment *getXAEnvironment(
   const string &dbname);
```

Parameter	Description
dbname	The database name; same as the optional dbname provided in the Open String (and used in connection to the Resource Manager).

13.14.21 releaseXAConnection()

Release/deallocate all resources allocated by the getXAConnection() method.

Syntax

```
void releaseXAConnection(
   Connection* conn);
```

Parameter	Description
conn	The connection returned by the getXAConnection() method.

13.14.22 releaseXAEnvironment()

Release/deallocate all resources allocated by the getXAEnvironment() method.

Syntax

```
void releaseXAEnvironment(
   Environment* env);
```

Parameter	Description
env	The environment returned by the getXAEnvironment() method.

13.14.23 setCacheMaxSize()

Sets the maximum size for the client-side object cache as a percentage of the optimal size. The default value is 10%.

Syntax

```
void setCacheMaxSize(
   unsigned int maxSize);
```

Parameter	Description
maxSize	The value of the maximum size, as a percentage.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.14.24 setCacheOptSize()

Sets the optimal size for the client-side object cache in bytes. The default value is 8MB.

Syntax

```
void setCacheOptSize(
   unsigned int optSize);
```

Parameter	Description
optSize	The value of the optimal size, in bytes.

13.14.25 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

```
void setCacheSortedFlush(
   bool flag);
```

Parameter	Description
flag	FALSE (default): no sorting; TRUE: sorting in table order

13.14.26 setLDAPAdminContext()

Sets the administrative context of the client. This is usually the root of the Oracle RDBMS LDAP schema in the LDAP server.

Syntax

```
void setLDAPAdminContext(
   const string &ctx);
```

Parameter	Description
ctx	The client context

13.14.27 setLDAPAuthentication()

Specifies the authentication mode. Currently the only supported value is 0x1: Simple authentication; username/password authentication.

Syntax

```
void setLDAPAuthentication(
   unsigned int mode);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
mode	The authentication mode

13.14.28 setLDAPHostAndPort()

Specifies the host on which the LDAP server is running, and the port on which it is listening for requests.

Syntax

```
void setLDAPHostAndPort(
   const string &host,
   unsigned int port);
```

Parameter	Description
host	The host for LDAP
port	The port for LDAP

13.14.29 setLDAPLoginNameAndPassword()

Specifies the login distinguished name and password used when connecting to an LDAP server.

Syntax

```
void setLDAPLoginNameAndPassword(
   const string &login,
   const &passwd);
```

Parameter	Description
login	The login name
passwd	The login password

13.14.30 setNLSLanguage()

Specifies the NLS Language for the current environment. The setting is effective for the connections created after this method has been called. The setting overrides the value set through the process environment variable NLS_LANG.

Syntax

```
void setNLSLanguage(
   const string &lang);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
lang	The language of the current environment

13.14.31 setNLSTerritory()

Specifies the NLS Territory for the current environment. The setting is effective for the connections created after this method has been called. The setting overrides the value set through the process environment variable NLS_LANG.

Syntax

```
void setNLSTerritory(
   const string &Terr);
```

Parameter	Description
Terr	The territory of the current environment

13.14.32 terminateConnection()

Closes the connection to the environment, and frees all related system resources.

Syntax

```
void terminateConnection(
   Connection *connection);
```

Parameter	Description
connection	A pointer to the connection instance to be terminated.

13.14.33 terminateConnectionPool()

Closes the connections in the connection pool, and frees all related system resources.

Syntax

```
void terminateConnectionPool(
   ConnectionPool *poolPointer);
```

Parameter	Description
poolPointer	A pointer to the connection pool instance to be terminated.

13.14.34 terminateEnvironment()

Closes the environment, and frees all related system resources.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void terminateEnvironment(
   Environment *env);
```

Parameter	Description
env	Environment to be closed.

13.14.35 terminateStatelessConnectionPool()

Destroys the specified StatelessConnectionPool.

Syntax

```
void termimnateStatelessConnectionPool(
   StatelessConnectionPool* poolPointer,
   StatelessConnectionPool::DestroyMode mode=StatelessConnectionPool::DEFAULT);
```

Parameter	Description
poolPointer	The StatelessConnectionPool to be destroyed.
mode	DestroyMode as defined Table 13-41: DEFAULT or SPF_FORCE.

13.15 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

Field	Type	Description
day	int	Day component. Valid values are -10^9 through 10^9.
hour	int	Hour component. Valid values are -23 through 23.
minute	int	Minute component. Valid values are -59 through 59.
second	int	Second component. Valid values are -59 through 59.
fs	int	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().

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.

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

// Create a NULL day-second interval
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
IntervalDS ds;
if(ds.isNull())
   cout << "\n 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:

```
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

Method	Summary
IntervalDS()	`IntervalDS` class constructor.
fromText()	Returns an `IntervalDS` converted from a `string`.
fromUText()	Returns an `IntervalDS` converted from a `UString`.
getDay()	Returns day interval values.
getFracSec()	Returns fractional second interval values.
getFracSec()	Returns hour interval values.
getMinute()	Returns minute interval values.
getSecond()	Returns second interval values.
isNull()	Returns `true` if `IntervalDS` is `NULL`, `false` otherwise.
operator*()	Returns the product of two `IntervalDS` values.
operator*=()	Multiplication assignment.
operator=()	Simple assignment.
operator==()	Checks if `a` and `b` are equal.
operator!=()	Checks if `a` and `b` are not equal.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-20 (Cont.) Summary of IntervalDS Methods

Method	Summary
operator/()	Returns an `IntervalDS` with value `(a / b)`.
operator/=()	Division assignment.
operator>()	Checks if `a` is greater than `b`
operator>=()	Checks if `a` is greater than or equal to `b`.
operator<()	Checks if `a` is less than `b`.
operator<=()	Checks if `a` is less than or equal to `b`.
operator-()	Returns an `IntervalDS` with value `(a - b)`.
operator-=()	Subtraction assignment.
operator+()	Returns the sum of two `IntervalDS` values.
operator+=()	Addition assignment.
set()	Sets day-second interval.
setNull()	Sets day-second interval to `NULL`.
toText()	Converts to a `string` representation for the interval.
toUText()	Converts to a `UString` representation for the interval.

13.15.1 IntervalDS()

`IntervalDS` class constructor.

Syntax	Description
`IntervalDS();`	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().

`IntervalDS(`
`  const Environment *env,`
`  int day = 0,`
`  int hour = 0,`
`  int minute = 0,`
`  int second = 0,`
`  int fs = 0);`
	Constructs an `IntervalDS` object within a specified `Environment`.

`IntervalDS(` `  const IntervalDS &src);`	Constructs an `IntervalYM` object from `src`.

Parameter	Description
`env`	The `Environment`.
`day`	The `day` field of `IntervalDS`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
hour	The hour field of IntervalDS.
minute	The minute field of IntervalDS.
second	The second field of IntervalDS.
fs	The fs field of IntervalDS.
src	The source that the IntervalDS object is copied from.

13.15.2 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

```
void fromText(
   const string &inpstr,
   const string &nlsParam = "",
   const Environment *env = NULL);
```

Parameter	Description
inpstr	Input string representing a day second interval of the form 'days hours:minutes:seconds', for example, '10 20:14:10.2'
nlsParam	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.
env	Environment whose NLS parameters are used.

13.15.3 fromUText()

Creates the interval from the UString specified.

Syntax

```
void fromUText(
   const UString &inpstr,
   const Environment *env=NULL );
```

Parameter	Description
inpstr	Input UString representing a day second interval of the form 'days hours:minutes:seconds', for example, '10 20:14:10.2'

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
env	The Environment.

13.15.4 getDay()

Returns the day component of the interval.

Syntax

```
int getDay() const;
```

13.15.5 getFracSec()

Returns the fractional second component of the interval.

Syntax

```
int getFracSec() const;
```

13.15.6 getHour()

Returns the hour component of the interval.

Syntax

```
int getHour() const;
```

13.15.7 getMinute()

Returns the minute component of this interval.

Syntax

```
int getMinute() const;
```

13.15.8 getSecond()

Returns the seconds component of this interval.

Syntax

```
int getSecond() const;
```

13.15.9 isNull()

Tests whether the interval is NULL. If the interval is NULL then TRUE is returned; otherwise, FALSE is returned.

Syntax

```
bool isNull() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.15.10 operator*()

Multiplies an interval by a specified value and returns the result.

Syntax

```
const IntervalDS operator*(
   const IntervalDS &interval,
   const Number &val);
```

Parameter	Description
`interval`	Interval to be multiplied.
`val`	Value by which interval is to be multiplied.

13.15.11 operator*=()

Assigns the product of `IntervalDS` and a to `IntervalDS`.

Syntax

```
IntervalDS& operator*=(
   const IntervalDS &factor);
```

Parameter	Description
`factor`	A day second interval.

13.15.12 operator=()

Assigns the specified value to the interval.

Syntax

```
IntervalDS& operator=(
   const IntervalDS &src);
```

Parameter	Description
`src`	Value to be assigned.

13.15.13 operator==()

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool operator==(
   const IntervalDS &first,
   const IntervalDS &second);
```

Parameter	Description
first	The first interval to be compared.
second	The second interval to be compared.

13.15.14 operator!=()

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

```
bool operator!=(
   const IntervalDS &first,
   const IntervalDS &second);
```

Parameter	Description
first	The first interval to be compared.
second	The second interval to be compared.

13.15.15 operator/()

Returns the result of dividing an interval by a constant value.

Syntax

```
const IntervalDS operator/(
   const IntervalDS &dividend,
   const Number &factor);
```

Parameter	Description
dividend	The interval to be divided.
factor	Value by which interval is to be divided.

13.15.16 operator/=()

Assigns the quotient of IntervalDS and val to IntervalDS.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
IntervalDS& operator/=(
   const IntervalDS &factor);
```

Parameter	Description
`factor`	A day second interval.

13.15.17 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

```
bool operator>(
   const IntervalDS &first,
   const IntervalDS &second);
```

Parameter	Description
`first`	The first interval to be compared.
`second`	The second interval to be compared.

13.15.18 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

```
bool operator>=(
   const IntervalDS &first,
   const IntervalDS &first);
```

Parameter	Description
`first`	The first interval to be compared.
`second`	The second interval to be compared.

13.15.19 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool operator<(
   const IntervalDS &first,
   const IntervalDS &second);
```

Parameter	Description
`first`	The first interval to be compared.
`second`	The second interval to be compared.

13.15.20 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

```
bool operator<=(
   const IntervalDS &first,
   const IntervalDS &second);
```

Parameter	Description
`first`	The first interval to be compared.
`second`	The second interval to be compared.

13.15.21 operator-()

Returns the difference between the intervals `first` and `second`.

Syntax

```
const IntervalDS operator-(
   const IntervalDS &first,
   const IntervalDS &second);
```

Parameter	Description
`first`	The first interval to be compared.
`second`	The second interval to be compared.

13.15.22 operator-=()

Assigns the difference between `IntervalDS` and `val` to `IntervalDS`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
IntervalDS& operator-=(
   const IntervalDS &val);
```

Parameter	Description
`val`	A day second interval.

13.15.23 operator+()

Returns the sum of the intervals specified.

Syntax

```
const IntervalDS operator+(
   const IntervalDS &first,
   const IntervalDS &second);
```

Parameter	Description
`first`	The first interval to be compared.
`second`	The second interval to be compared.

13.15.24 operator+=()

Assigns the sum of `IntervalDS` and `val` to `IntervalDS`.

Syntax

```
IntervalDS& operator+=(
   const IntervalDS &val);
```

Parameter	Description
`val`	A day second interval.

13.15.25 set()

Sets the interval to the values specified.

Syntax

```
void set(
   int day,
   int hour,
   int minute,
   int second,
   int fracsec);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
`day`	Day component.
`hour`	Hour component.
`min`	Minute component.
`second`	Second component.
`fracsec`	Fractional second component.

13.15.26 setNull()

Sets the `IntervalDS` to `NULL`.

Syntax

```
void setNull();
```

13.15.27 toText()

Converts to a `string` representation for the interval.

Syntax

```
string toText(
   unsigned int lfprec,
   unsigned int fsprec,
   const string &nlsParam = "") const;
```

Parameter	Description
`lfprec`	Leading field precision.
`fsprec`	Fractional second precision.
`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`.

13.15.28 toUText()

Converts to a `UString` representation for the interval.

Syntax

```
UString toUText(
   unsigned int lfprec,
   unsigned int fsprec) cosnt;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
`lfprec`	Leading field precision.
`fsprec`	Fractional second precision.

13.16 IntervalYM Class

`IntervalYM` supports the SQL standard data type Year-Month Interval.

Leading field precision is determined by number of decimal digits on input.

Table 13-21 Fields of IntervalYM Class

Field	Type	Description
`year`	`int`	Year component. Valid values are `-10^9` through `10^9`.
`month`	`int`	Month component. Valid values are `-11` through `11`.

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:

```
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 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.

```
//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
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
IntervalYM mulYm1 = ym1 * Number(env, 10);    //call * operator
if(ym1<mulYm1)    //comparison
   .
   .
string strym = ym1.toText(3);    //3 is the leading field precision
```

Table 13-22 Summary of IntervalYM Methods

Method	Summary
IntervalYM()	`IntervalYM` class constructor.
fromText()	Converts a `string` into an `IntervalYM`.
fromUText()	Converts a `UString` into an `IntervalYM`.
getMonth()	Returns month interval value.
getYear()	Returns year interval value.
isNull()	Checks if the interval is `NULL`.
operator*()	Returns the product of two `IntervalYM` values.
operator*=()	Multiplication assignment.
operator=()	Simple assignment.
operator==()	Checks if `a` and `b` are equal.
operator!=()	Checks if `a` and `b` are not equal.
operator/()	Returns an `interval` with value `(a/b)`.
operator/=()	Division assignment.
operator>()	Checks if `a` is greater than `b`.
operator>=()	Checks if `a` is greater than or equal to `b`.
operator<()	Checks if `a` is less than `b`.
operator<=()	Checks if `a` is less than or equal to `b`.
operator-()	Returns an `interval` with value `(a - b)`.
operator-=()	Subtraction assignment.
operator+()	Returns the sum of two `IntervalYM` values.
operator+=()	Addition assignment.
set()	Sets the interval to the values specified.
setNull()	Sets the interval to `NULL`.
toText()	Converts to a `string` representation of the interval.
toUText()	Converts to a `UString` representation of the interval.

13.16.1 IntervalYM()

IntervalYM class constructor.

Syntax	Description
`IntervalYM();`	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().

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`IntervalYM( const Environment *env, int year = 0, int month = 0);`	Creates an `IntervalYM` object within the specified `Environment`.
`IntervalDS( const IntervalYM &src);`	Copy constructor.

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.

13.16.2 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

```
void fromText(
   const string &inpStr,
   const string &nlsParam = "",
   const Environment *env = NULL);
```

Parameter	Description
`inpStr`	Input string representing a year month interval of the form 'year-month'.
`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`.
`env`	Environment whose NLS parameters are used.

13.16.3 fromUText()

Creates the interval from the `UString` specified.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void fromUText(
   const UString &inpStr,
   const Environment *env=NULL );
```

Parameter	Description
`inpStr`	Input `UString` representing a year month interval of the form 'year-month'.
`env`	The `Environment`.

13.16.4 getMonth()

This method returns the month component of the interval.

Syntax

```
int getMonth() const;
```

13.16.5 getYear()

This method returns the year component of the interval.

Syntax

```
int getYear() const;
```

13.16.6 isNull()

This method tests whether the interval is `NULL`. If the interval is `NULL` then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

13.16.7 operator*()

This method multiplies the interval by a factor and returns the result.

Syntax

```
const IntervalYM operator*(
   const IntervalDS &interval
   const Number &val);
```

Parameter	Description
`interval`	Interval to be multiplied.
`val`	Value by which interval is to be multiplied.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.16.8 operator*=()

This method multiplies the interval by a specified value.

Syntax

```
IntervalYM& operator*=(
   const Number &factor);
```

Parameter	Description
`factor`	Value to be multiplied.

13.16.9 operator=()

This method assigns the specified value to the interval.

Syntax

```
IntervalYM& operator=(
   const IntervalYM &src);
```

Parameter	Description
`src`	Value to be assigned.

13.16.10 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

```
bool operator==(
   const IntervalYM &first,
   const IntervalYM &second);
```

Parameter	Description
`first`	The first interval to be compared.
`second`	The second interval to be compared.

13.16.11 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool operator!=(
   const IntervalYM &first,
   const IntervalYM &second);
```

Parameter	Description
first	The first interval to be compared.
second	The second interval to be compared.

13.16.12 operator/()

This method returns the result of dividing the interval by a factor.

Syntax

```
const IntervalYM operator/(
   const IntervalYM &dividend,
   const Number &factor);
```

Parameter	Description
dividend	The interval to be divided.
factor	Value by which interval is to be divided.

13.16.13 operator/=()

This method divides the interval by a factor.

Syntax

```
IntervalYM& operator/=(
   const Number &factor);
```

Parameter	Description
factor	A day second interval.

13.16.14 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

```
bool operator>(
   const IntervalYM &first,
   const IntervalYM &second);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
first	The first interval to be compared.
second	The second interval to be compared.

13.16.15 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

```
bool operator>=(
   const IntervalYM &first,
   const IntervalYM &second);
```

Parameter	Description
first	The first interval to be compared.
second	The second interval to be compared.

13.16.16 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

```
bool operator<(
   const IntervalYM &first,
   const IntervalYM &second);
```

Parameter	Description
first	The first interval to be compared.
second	The second interval to be compared.

13.16.17 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool operator<=(
   const IntervalYM &first,
   const IntervalYM &second);
```

Parameter	Description
first	The first interval to be compared.
second	The second interval to be compared.

13.16.18 operator-()

This method returns the difference between the intervals specified.

Syntax

```
const IntervalYM operator-(
   const IntervalYM &first,
   const IntervalYM &second);
```

Parameter	Description
first	The first interval to be compared.
second	The second interval to be compared.

13.16.19 operator-=()

This method computes the difference between itself and another interval.

Syntax

```
IntervalYM& operator-=(
   const IntervalYM &val);
```

Parameter	Description
val	A day second interval.

13.16.20 operator+()

This method returns the sum of the intervals specified.

Syntax

```
const IntervalYM operator+(
   const IntervalYM &first,
   const IntervalYM &second);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
first	The first interval to be compared.
second	The second interval to be compared.

13.16.21 operator+=()

This method assigns the sum of IntervalYM and val to IntervalYM.

Syntax

```
IntervalYM& operator+=(
   const IntervalYM &val);
```

Parameter	Description
val	A day second interval.

13.16.22 set()

This method sets the interval to the values specified.

Syntax

```
void set(
   int year,
   int month);
```

Parameter	Description
year	Year component. Valid values are -10^9 through 10^9.
month	Month component. Valid values are -11 through 11.

13.16.23 setNull()

This method sets the interval to NULL.

Syntax

```
void setNull();
```

13.16.24 toText()

This method returns the string representation of the interval.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
string toText(
   unsigned int lfprec,
   const string &nlsParam = "") const;
```

Parameter	Description
`lfprec`	Leading field precision.
`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`.

13.16.25 toUText()

Converts to a `UString` representation for the interval.

Syntax

```
UString toUText(
   unsigned int lfprec) cosnt;
```

Parameter	Description
`lfprec`	Leading field precision.

13.17 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

Method	Summary
Listener()	`Listener` class constructor.
getAgentList()	Retrieves the list of `Agents` for which the `Listener` provides its services.
getTimeOutForListen()	Retrieves the time out for a call.
listen()	Listens for `Messages` and returns the name of the `Agent` for whom a `Message` is intended.
setAgentList()	Specifies the list of `Agents` for which the `Listener` provides its services.
setTimeOutForListen()	Specifies the time out for a listen() call.

13.17.1 Listener()

`Listener` class constructor.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`Listener( const Connection* conn);`	Creates a `Listener` object.
`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
`conn`	The connection of the new `Listener` object.
`aglist`	The list of agents on behalf of which the `Listener` object waits on queues; clients of this `Listener`.
`waitTime`	The time to wait on queues for messages of interest for the clients; in seconds.

13.17.2 getAgentList()

Retrieves the list of `Agents` for which the `Listener` provides its services.

Syntax

```
vector<Agent> getAgentList() const;
```

13.17.3 getTimeOutForListen()

Retrieves the time out for a call.

Syntax

```
int getTimeOutForListen() const;
```

13.17.4 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

```
Agent listen();
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.17.5 setAgentList()

Specifies the list of `Agents` for which the `Listener` provides its services.

Syntax

```
void setAgentList(
   vector<Agent>& agList);
```

Parameter	Description
`agList`	The list of `Agents`.

13.17.6 setTimeOutForListen()

Specifies the time out for a listen() call.

Syntax

```
void setTimeOutForListen(
   int waitTime);
```

Parameter	Description
`waitTime`	The time interval, in seconds, during which the `Listener` is waiting for `Messages` at specified queues.

13.18 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

Method	Summary
put()	Adds a map entry for the type `type_name`.

13.18.1 put()

Adds a map entry for the type, `type_name`, that must be customized; you must implement the `type_name` C++ class.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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 this method after the environment is created passing the environment.

Syntax	Description
`void put( const string &schemaType, void *(*rSQL)(void *), void (*wSQL) (void *, void *));`	Registers a type and its corresponding C++ `readSQL` and `writeSQL` functions.
`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.
`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	Description
`schemaType`	The schema and typename, separated by ".", like `HR.TYPE1`
`schName`	Name of the schema
`typName`	Name of the type
`rDQL`	The `readSQL` function pointer of the C++ class that corresponds to the type
`wSQL`	The `writeSQL` function pointer of the C++ class that corresponds to the type

13.19 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 13-25 Enumerated Values Used by Message Class

Attribute	Options
`MessageState`	• `MSG_WAITING` indicates that the message delay time has not been reached • `MSG_READY` indicates that the message is ready to be processed • `MSG_PROCESSED` indicates that the message has been processed, and is being retained • `MSG_EXPIRED` indicates that the message has been moved to the exception queue.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-25 (Cont.) Enumerated Values Used by Message Class

Attribute	Options
PayloadType	• RAW • ANYDATA • OBJECT

Table 13-26 Summary of Message Methods

Method	Summary
Message()	Message class constructor.
getAnyData()	Retrieves AnyData payload of the message.
getAttemptsToDequeue()	Retrieves the number of attempts made to dequeue the message.
getBytes()	Retrieves Bytes payload of the message.
getCorrelationId()	Retrieves the identification string.
getDelay()	Retrieves delay with which message was enqueued.
getExceptionQueueName()	Retrieves name of queue to which Message is moved when it cannot be processed.
getExpiration()	Retrieves the expiration of the message.
getMessageEnqueuedTime()	Retrieves time at which message was enqueued.
getMessageState()	Retrieves state of the message at time of enqueuing.
getObject()	Retrieves object payload of the message.
getOriginalMessageId()	Retrieves the Id of the message that generated this message on the last queue.
getPayloadType()	Retrieves the type of the payload.
getPriority()	Retrieves the priority of the message.
getSenderId()	Retrieves the agent who send the Message.
isNull()	Tests whether the Message object is NULL.
operator=()	Assignment operator for Message.
setAnyData()	Specifies AnyData payload of the message.
setBytes()	Specifies Bytes payload of the message.
setCorrelationId()	Specifies the identification string.
setDelay()	Specifies the number of seconds to delay the enqueued Message.
setExceptionQueueName()	Specifies the name of the queue to which the Message object is moved if it cannot be precessed.
setExpiration()	Specifies the duration of time that Message can be dequeued before it expires.
setNull()	Sets the Message object to NULL.
setObject()	Specifies object payload of the message.
setOriginalMessageId()	Specifies id of last queue that generated the Message.
setPriority()	Specifies priority of the Message object.
setRecipientList()	Specifies the list of agents for whom the message is intended.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-26 (Cont.) Summary of Message Methods

Method	Summary
setSenderId()	Specifies the sender of the Message.

13.19.1 Message()

Message class constructor.

Syntax	Description
`Message( const Environment *env);`	Creates a Message object within the specified Environment.
`Message( const Message& mes);`	Copy constructor.

Parameter	Description
env	The environment of the Message.
mes	The original Message.

13.19.2 getAnyData()

Retrieves the AnyData payload of the Message.

Syntax

```
AnyData getAnyData() const;
```

13.19.3 getAttemptsToDequeue()

Retrieves the number of attempts made to dequeue the message. This property cannot be retrieved while enqueuing.

Syntax

```
int getAttemptsToDequeue() const;
```

13.19.4 getBytes()

Retrieves Bytes payload of the Message.

Syntax

```
Bytes getBytes() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.19.5 getCorrelationId()

Retrieves the identification string.

Syntax

```
string getCorrelationId() const;
```

13.19.6 getDelay()

Retrieves the delay (in seconds) with which the `Message` was enqueued.

Syntax

```
int getDelay() const;
```

13.19.7 getExceptionQueueName()

Retrieves the name of the queue to which the `Message` is moved, in cases when the `Message` cannot be processed successfully.

Syntax

```
string getExceptionQueueName() const;
```

13.19.8 getExpiration()

Retrieves the expiration time of the `Message` (in seconds). This is the duration for which the message is available for dequeuing.

Syntax

```
int getExpiration() const;
```

13.19.9 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;
```

13.19.10 getMessageState()

Retrieves the state of the message at the time of enqueuing. This parameter cannot be set an enqueuing time. `MessageState` is defined in Table 13-25.

Syntax

```
MessageState getMessageState() const;
```

13.19.11 getObject()

Retrieves object payload of the `Message`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
PObject* getObject();
```

13.19.12 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;
```

13.19.13 getPayloadType()

Retrieves the type of the payload, as defined for `PayloadType` in Table 13-25.

Syntax

```
PayloadType getPayloadType( ) const;
```

13.19.14 getPriority()

Retrieves the priority of the `Message`.

Syntax

```
int getPriority() const;
```

13.19.15 getSenderId()

Retrieves the agent who send the `Message`.

Syntax

```
Agent getSenderId() const;
```

13.19.16 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;
```

13.19.17 operator=()

Assignment operator for `Message`.

```
void operator=(
   const Message& mes);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
mes	Original message.

13.19.18 setAnyData()

Specifies AnyData payload of the Message.

Syntax

```
void setAnyData(
   const AnyData& anydata);
```

Parameter	Description
anydata	Data content of the Message.

13.19.19 setBytes()

Specifies Bytes payload of the Message.

Syntax

```
void setBytes(
   const Bytes& bytes);
```

Parameter	Description
bytes	Data content of the Message.

13.19.20 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

```
void setCorrelationId(
   const string& id);
```

Parameter	Description
id	The id; upper limit of 128 bytes.

13.19.21 setDelay()

Specifies the time (in seconds) to delay the enqueued Message. After the delay ends, the Message is available for dequeuing.

Copyright © 1999, 2025, Oracle and/or its affiliates.

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

```
void setDelay(
   int delay);
```

Parameter	Description
`delay`	The delay.

13.19.22 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

```
void setExceptionQueueName(
   const string& queue);
```

Parameter	Description
`queue`	The name of the exception queue.

13.19.23 setExpiration()

Specifies the duration time (in seconds) that the `Message` object is available for dequeuing. A `Message` expires after this time.

Syntax

```
void setExpiration(
   int exp);
```

Parameter	Description
`exp`	The duration of expiration.

13.19.24 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setNull();
```

13.19.25 setObject()

Specifies object payload of the `Message`.

Syntax

```
void setObject(
   PObject& pobj);
```

Parameter	Description
`pobj`	Content of the data

13.19.26 setOriginalMessageId()

Sets the Id of the last queue that generated the message, when a message is propagated from one queue to another.

Syntax

```
void setOriginalMessageId(
   const Bytes& queue);
```

Parameter	Description
`queue`	The last queue.

13.19.27 setPriority()

Specifies the priority of the `Message` object. This property is set during enqueuing time, and can be negative. Default is `0`.

Syntax

```
void setPriority(
   int priority);
```

Parameter	Description
`priority`	The priority of the `Message`.

13.19.28 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setRecipientList(
   vector<Agent>& agentList);
```

Parameter	Description
`agentList`	The list of `Agents`.

13.19.29 setSenderId()

Specifies the sender of the `Message`.

Syntax

```
void setSenderId(
   const Agent& sender);
```

Parameter	Description
`sender`	Sender id.

13.20 MetaData Class

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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-27 Enumerated Values Used by MetaData Class

Attribute	Options
ParamType	The parameter types for objects are: • PTYPE_ARG is the argument of a function or procedure. • PTYPE_COL is the column of a table or view. • PTYPE_DATABASE is the database. • PTYPE_FUNC is the function. • PTYPE_PKG is the package. • PTYPE_PROC is the procedure. • PTYPE_SCHEMA is the schema. • PTYPE_SEQ is the sequence. • PTYPE_SYN is the synonym. • PTYPE_TABLE is the table. • PTYPE_TYPE is the type. • PTYPE_TYPE_ARG is the argument of a type method. • PTYPE_TYPE_ATTR is the attribute of a type. • PTYPE_TYPE_COLL is the collection type information. • PTYPE_TYPE_METHOD is the method of a type. • PTYPE_TYPE_RESULT is the results of a method. • PTYPE_UNK is the object of an unknown type. • PTYPE_VIEW is the view.
AttrId common to all parameters	Attributes of all parameters: • ATTR_OBJ_ID is the object or schema id. • ATTR_OBJ_NAME is either the database name, or the object name in a schema. • ATTR_OBJ_SCHEMA is the name of the schema describing the object. • ATTR_PTYPE is the type of information described by a parameter, ParamType • ATTR_TIMESTAMP is the timestamp of an object.
AttrId for Tables and Views	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
AttrId for Tables only	Parameters for a tables only (ParamType of PTYPE_TABLE): • 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

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-27 (Cont.) Enumerated Values Used by MetaData Class

Attribute	Options
`AttrId` for Functions and Procedures	Parameters for functions and procedures (`ParamType` of `PTYPE_FUNC` and `PTYPE_PROC`, respectively): • `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
`AttrId` for Packages	Parameters for packages (`Paramtype` of `PTYPE_PKG`): • `ATTR_LIST_SUBPROGRAMS` indicates the subprogram list • `ATTR_IS_INVOKER_RIGHTS` indicates the procedure or function has invoker's rights
`AttrId` for Types	Parameter is for types (`ParamType` of `PTYPE_TYPE`): • `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. • `ATTR_TYPECODE` indicates the data type code • `ATTR_COLLECTION_TYPECODE` indicates the typecode of collection, if type is collection • `ATTR_IS_INCOMPLETE_TYPE` indicates that this is an incomplete type • `ATTR_IS_SYSTEM_TYPE` indicates that this is a system generated type • `ATTR_IS_PREDEFINED_TYPE` indicates that this is a predefined type • `ATTR_IS_TRANSIENT_TYPE` indicates that this is a transient type • `ATTR_IS_SYSTEM_GENERATED_TYPE` indicates that this is a system generated type • `ATTR_HAS_NESTED_TABLE` indicates that this type contains a nested table attribute • `ATTR_HAS_LOB` indicates that this type contains a `LOB` attribute • `ATTR_HAS_FILE` indicates that this type contains a `BFILE` attribute • `ATTR_COLLECTION_ELEMENT` indicates a reference to a collection element • `ATTR_NUM_TYPE_ATTRS` indicates the number of type attributes • `ATTR_LIST_TYPE_ATTRS` indicates the list of type attributes • `ATTR_NUM_TYPE_METHODS` indicates the number of type methods • `ATTR_LIST_TYPE_METHODS` indicates the list of type methods • `ATTR_MAP_METHOD` indicates the map method of the type • `ATTR_ORDER_METHOD` indicates the order method of the type • `ATTR_IS_INVOKER_RIGHTS` indicates the type has invoker's rights • `ATTR_NAME` indicates the type attribute name • `ATTR_SCHEMA_NAME` indicates the schema where the type is created • `ATTR_IS_FINAL_TYPE` indicates this is a final type • `ATTR_IS_INSTANTIABLE_TYPE` indicates this is an instantiable type • `ATTR_IS_SUBTYPE` indicates this is a subtype • `ATTR_SUPERTYPE_SCHEMA_NAME` indicates the name of the schema that contains the supertype • `ATTR_SUPERTYPE_NAME` indicates the name of the supertype

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-27 (Cont.) Enumerated Values Used by MetaData Class

Attribute	Options
`AttrId` for Type Attributes	Parameter is for attributes of types (`ParamType` of `PTYPE_TYPE_ATTR`): • `ATTR_DATA_SIZE` indicates the maximum size of the type attribute • `ATTR_TYPECODE` indicates the data type code • `ATTR_DATA_TYPE` indicates the data type of the type attribute • `ATTR_NAME` indicates the name of the procedure or function • `ATTR_PRECISION` indicates the precision of numeric type attributes. • `ATTR_SCALE` indicates the scale of the numeric type attributes • `ATTR_TYPE_NAME` indicates a type name • `ATTR_SCHEMA_NAME` indicates the name of the schema where the type has been created • `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. • `ATTR_CHARSET_ID` indicates the characterset ID • `ATTR_CHARSET_FORM` indicates the characterset form • `ATTR_FSPRECISION` indicates the fractional seconds precision of a `Timestamp`, `IntervalDS` or `IntervalYM` • `ATTR_LFPRECISION` indicates the leading field precision of an `IntervalDS` or `IntervalYM`
`AttrId` for Type Methods	Parameter is for methods of types (`ParamType` of `PTYPE_METHOD`): • `ATTR_NAME` indicates the name of the procedure or function • `ATTR_ENCAPSULATION` indicates the method's level of encapsulation • `ATTR_LIST_ARGUMENTS` indicates the argument list • `ATTR_IS_CONSTRUCTOR` indicates the method is a constructor • `ATTR_IS_DESTRUCTOR` indicates the method is a destructor • `ATTR_IS_OPERATOR` indicates the method is an operator • `ATTR_IS_SELFISH` indicates the method is selfish • `ATTR_IS_MAP` indicates the method is a map method • `ATTR_IS_ORDER` indicates the method is an order method • `ATTR_IS_RNDS` indicates that the method is in "read no data" state • `ATTR_IS_RNPS` indicates that the method is in a "read no process" state • `ATTR_IS_WNDS` indicates that the method is in "write no data" state • `ATTR_IS_WNPS` indicates that the method is in "write no process" state • `ATTR_IS_FINAL_METHOD` indicates that this is a final method • `ATTR_IS_INSTANTIABLE_METHOD` indicates that this is an instantiable method • `ATTR_IS_OVERRIDING_METHOD` indicates that this is an overriding method

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-27 (Cont.) Enumerated Values Used by MetaData Class

Attribute	Options
`AttrId` for Collections	Parameter is for collections (`ParamType` of `PTYPE_TYPE_COLL`): • `ATTR_DATA_SIZE` indicates ... • `ATTR_TYPECODE` indicates ... • `ATTR_DATA_TYPE` indicates the data type of the type attribute • `ATTR_NUM_ELEMS` indicates the number of elements in a collection • `ATTR_NAME` indicates the name of the type attribute • `ATTR_PRECISION` indicates the precision of a numeric attribute • `ATTR_SCALE` indicates the scale of a numeric attribute • `ATTR_TYPE_NAME` indicates the type name • `ATTR_SCHEMA_NAME` indicates the schema where the type has been created • `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. • `ATTR_CHARSET_ID` indicates the characterset id • `ATTR_CHARSET_FORM` indicates the characterset form • `ATTR_IS_IDENTITY` indicates that the column may be auto-incremented
`AttrId` for Synonyms	Parameter is for synonyms (`ParamType` of `PTYPE_SYN`): • `ATTR_OBJID` indicates the object id • `ATTR_SCHEMA_NAME` indicates the schema name of the synonym translation • `ATTR_NAME` indicates a `NULL`-terminated object name of the synonym translation • `ATTR_LINK` indicates a `NULL-`terminated database link name of the synonym installation
`AttrId` for Sequences	Parameter is for sequences (`ParamType` of `PTYPE_SEQ`): • `ATTR_OBJID` indicates the object id • `ATTR_MIN` indicates the minimum value • `ATTR_MAX` indicates the maximum value • `ATTR_INCR` indicates the increment • `ATTR_CACHE` indicates the number of sequence numbers cached; 0 if the sequence is not cached • `ATTR_ORDER` indicates whether the sequence is ordered • `ATTR_HW_MARK` indicates the "high-water mark"

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-27 (Cont.) Enumerated Values Used by MetaData Class

Attribute	Options
`AttrId` for Columns	Parameter is for columns of tables or views (`ParamType` of `PTYPE_COL`): • `ATTR_CHAR_USED` indicates the type of length semantics of the column. `0` means byte-length semantics and `1` means character-length semantics. • `ATTR_CHAR_SIZE` indicates the column character length, or number of characters allowed in a column • `ATTR_DATA_SIZE` indicates the maximum size of a column, or number of bytes allowed in a column • `ATTR_DATA_TYPE` indicates the data type of the column • `ATTR_NAME` indicates the column name • `ATTR_PRECISION` indicates the precision of numeric columns • `ATTR_SCALE` indicates the scale of numeric columns • `ATTR_IS_NULL` indicates `0` if `NULL` values are not permitted for the column • `ATTR_TYPE_NAME` indicates a type name • `ATTR_SCHEMA_NAME` indicates the schema where the type was created • `ATTR_REF_TDO` indicates the `REF` for the type, if the column is of object type • `ATTR_CHARSET_ID` indicates the characterset ID • `ATTR_CHARSET_FORM` indicates the characterset form
`AttrId` for Arguments and Results	Parameter for arguments of a procedure or function (`PTYPE_ARG`), a method (`PTYPE_TYPE_ARG`), or a result (`PTYPE_TYPE_RESULT`) • `ATTR_NAME` indicates the argument name • `ATTR_POSITION` indicates the position of the argument in the list • `ATTR_TYPECODE` indicates the typecode • `ATTR_DATA_TYPE` indicates the data type • `ATTR_DATA_SIZE` indicates the size of the data type • `ATTR_PRECISION` indicates the precision of a numeric argument • `ATTR_SCALE` indicates the scale of a numeric argument • `ATTR_LEVEL` indicates the data type level • `ATTR_HAS_DEFAULT` indicates whether an argument has a default • `ATTR_LIST_ARGUMENTS` indicates the list of arguments at the next level, for records or table types • `ATTR_IOMODE` indicates the argument mode: `0` for `IN`, `1` for `OUT`, `2` for `IN/OUT` • `ATTR_RADIX` indicates the radix of a number type • `ATTR_IS_NULL` indicates `0` if `NULL` values are not permitted • `ATTR_TYPE_NAME` indicates the type name • `ATTR_SCHEMA_NAME` indicates the schema name where the type was created • `ATTR_SUB_NAME` indicates the type name for package local types • `ATTR_LINK` indicates a `NULL`-terminated database link name where the type is defined, for package local types when the package is remote • `ATTR_REF_TDO` is the `REF` to the `TDO` of the type if the argument is an object • `ATTR_CHARSET_ID` indicates the characterset ID • `ATTR_CHARSET_FORM` indicates the characterset form
`AttrId` for Schemas	Parameter is for schemas (`ParamType` of `PTYPE_SCHEMA`): • `ATTR_LIST_OBJECTS` indicates the list of objects in the schema

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-27 (Cont.) Enumerated Values Used by MetaData Class

Attribute	Options
`AttrId` for Lists	Parameter is for list of columns, arguments or subprograms: • `ATTR_LIST_COLUMNS` indicates a column list • `ATTR_LIST_ARGUMENTS` indicates a procedure or function argument list • `ATTR_LIST_SUBPROGRAMS` indicates a subprogram list • `ATTR_LIST_TYPE_ATTRIBS` indicates a type attribute list • `ATTR_TYPE_METHODS` indicates a type method list • `ATTR_TYPE_OBJECTS` indicates a list of objects in a schema • `ATTR_LIST_SCHEMAS` indicates a list of schemas in a database
`AttrId` for Databases	Parameter is for list of columns, arguments or subprograms (`ParamType` of `PTYPE_DATABASE`): • `ATTR_VERSION` indicates the database version • `ATTR_CHARSET_ID` indicates the characterset ID of the database • `ATTR_NCHARSET_ID` indicates the national characterset of the database • `ATTR_LIST_SCHEMAS` indicates the list of schemas, PTYPE_SCHEMA • `ATTR_MAX_PROC_LEN` indicates the maximum length of a procedure name • `ATTR_MAX_COLUMN_LEN` indicates the maximum length of a column name • `ATTR_CURSOR_COMMIT_BEHAVIOR` indicates how a commit affects cursors and prepared statements. Values can be CURSOR_OPEN and CURSER_CLOSED, as defined for attribute `AttrValues` • `ATTR_MAX_CATALOG_NAMELEN` indicates the maximum length of a database (catalog) name • `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` • `ATTR_SAVEPOINT_SUPPORT` indicates whether the database supports savepoints. Values can be SP_SUPPORTED and SP_UNSUPPORTED, as defined for attribute `AttrValues` • `ATTR_NOWAIT_SUPPORT` indicates whether the database supports the "no wait" condition. Values can be NW_SUPPORTED and NW_UNSUPPORTED, as defined for attribute `AttrValues` • `ATTR_AUTOCOMMIT_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` • `ATTR_LOCKING_MODE` indicates the locking mode for the database. Values can be LOCK_IMMEDIATE and LOCK_DELAYED, as defined for attribute `AttrValues`

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-27 (Cont.) Enumerated Values Used by MetaData Class

Attribute	Options
`AttrValues`	Attribute values are returned on executing a `getxxx()` method and passing in an attribute, for which these are the results: • `DURATION_SESSION` is the duration of a temporary table: session. • `DURATION_TRANS` is the duration of a temporary table: transaction. • `DURATION_NULL` is the duration of a temporary table: table not temporary. • `TYPEENCAP_PRIVATE` is the encapsulation level of the method: private. • `TYPEENCAP_PUBLIC` is the encapsulation level of the method: public. • `TYPEPARAM_IN` is the argument mode: `IN`. • `TYPEPARAM_OUT` is the argument mode: `OUT`. • `TYPEPARAM_INOUT` is the argument mode: `IN/OUT`. • `CURSOR_OPEN` is the effect of `COMMIT` operation on cursors and prepared statements in the database: preserve cursor state as before the `COMMIT` operation. • `CURSER_CLOSED` is the effect of `COMMIT` operation on cursors and prepared statements in the database: cursors are closed on `COMMIT`, but the application can still rerun the statement without preparing it again. • `CL_START` is the position of the catalog in a qualified table: start. • `CL_END` is the position of the catalog in a qualified table: end. • `SP_SUPPORTED` is the database supports savepoints. • `SP_UNSUPPORTED` is the database does not support savepoints. • `NW_SUPPORTED` is the database supports nowait clause. • `NW_UNSUPPORTED` is the database does not supports nowait clause. • `AC_DDL` is the autocommit mode required for DDL statements. • `NO_AC_DDL` is the autocommit mode not required for DDL statements. • `LOCK_IMMEDIATE` is the locking mode for the database: immediate. • `LOCK_DELAYED` is the locking mode for the database: delayed.
`ColumnAttrId`	Attributes for column identity enable automatic increment support. Possible values are: • `ATTR_COL_IS_IDENTITY` is `true` when column is an identity column. • `ATTR_COL_IS_GEN_ALWAYS` is `true` when the column is always generated. • `ATTR_COL_IS_GEN_BY_DEF_ON_NULL` is `true` when the identity column is generated by default on `null`.

Table 13-28 Summary of MetaData Methods

Method	Description
MetaData()	`MetaData` class constructor.
getAttributeCount()	Gets the count of the attribute as a `MetaData` object
getAttributeId()	Gets the ID of the specified attribute
getAttributeType()	Gets the type of the specified attribute.
getBoolean()	Gets the value of the attribute as a C++ `boolean`.
getInt()	Gets the value of the attribute as a C++ `int`.
getMetaData()	Gets the value of the attribute as a `MetaData` object
getNumber()	Returns the specified attribute as a `Number` object.
getRef()	Gets the value of the attribute as a `Ref<T>`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-28 (Cont.) Summary of MetaData Methods

Method	Description
getString()	Gets the value of the attribute as a string.
getTimeStamp()	Gets the value of the attribute as a `Timestamp` object
getUInt()	Gets the value of the attribute as a C++ `unsigned int`.
getUString()	Returns the value of the attribute as a `UString` in the character set associated with the metadata.
getVector()	Gets the value of the attribute as an C++ vector.
operator=()	Assigns one metadata object to another.

13.20.1 MetaData()

`MetaData` class constructor.

Syntax

```
MetaData(
   const MetaData &omd);
```

Parameter	Description
`cmd`	The source that the `MetaData` object is copied from.

13.20.2 getAttributeCount()

This method returns the number of attributes related to the metadata object.

Syntax

```
unsigned int getAttributeCount() const;
```

13.20.3 getAttributeId()

This method returns the attribute ID, such as `ATTR_NUM_COLS`, of the attribute represented by the attribute number specified.

Syntax

```
AttrId getAttributeId(
   unsigned int attributeNum) const;
```

Parameter	Description
`attributeNum`	The number of the attribute for which the attribute ID is to be returned.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.20.4 getAttributeType()

This method returns the attribute type, such as `NUMBER` or `INT`, of the attribute represented by attribute number specified.

Syntax

```
Type getAttributeType(
   unsigned int attributeNum) const;
```

Parameter	Description
`attributeNum`	The number of the attribute for which the attribute type is to be returned.

13.20.5 getBoolean()

This method returns the value of the attribute as a C++ `boolean`. If the value is a SQL `NULL`, the result is `FALSE`. The overloaded version returns the value of the column attribute.

Syntax	Description
`bool getBoolean(` `  MetaData::AttrId attributeId) const;`	Returns the value of the attribute.
`bool getBoolean(` `  MetaData::ColumnAttrId colAttributeId) const;`	Returns the value of the column attribute

Parameter	Description
`attributeId`	The attribute ID
`colAttributeId`	The column attribute ID

13.20.6 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;
```

Parameter	Description
`attributeId`	The attribute ID

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.20.7 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;
```

Parameter	Description
`attributeId`	The attribute ID

13.20.8 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

```
Number getNumber(
   MetaData::AttrId attributeId) const;
```

Parameter	Description
`attributeId`	The attribute ID

13.20.9 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

```
RefAny getRef(
   MetaData::AttrId attributeId) const;
```

Parameter	Description
`attributeId`	The attribute ID

13.20.10 getString()

This method returns the value of the attribute as a string. If the value is SQL `NULL`, the result is `NULL`.

Starting with Oracle AI Database 26ai, the following two new attributes are introduced: `MetaData::ATTR_DOMAIN_NAME` and `MetaData::ATTR_DOMAIN_SCHEMA` for class `MetaData` to represent domain name and domain schema respectively.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
string getString(
   MetaData::AttrId attributeId) const;
```

Parameter	Description
`attributeId`	The attribute ID

13.20.11 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

```
Timestamp getTimestamp(
   MetaData::AttrId attributeId) const;
```

Parameter	Description
`attributeId`	The attribute ID

13.20.12 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

```
unsigned int getUInt(
   MetaData::AttrId attributeId) const;
```

Parameter	Description
`attributeId`	The attribute ID

13.20.13 getUString()

Returns the value of an attribute as a `UString` in the character set associated with the metadata.

Syntax

```
UString getUString(
   MetaData::AttrId attributeId) const;
```

Parameter	Description
`attributeId`	The attribute ID

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.20.14 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

```
vector<MetaData> getVector(
   MetaData::AttrId attributeId) const;
```

Parameter	Description
`attributeId`	The attribute ID

13.20.15 operator=()

This method assigns one `MetaData` object to another. This increments the reference count of the `MetaData` object that is assigned.

Syntax

```
void operator=(
   const MetaData &omd);
```

Parameter	Description
`cmd`	MetaData object to be assigned

13.21 NotifyResult Class

A NotifyResult object holds the notification information in the Database 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

Method	Summary
getConsumerName()	Returns the name of the notification consumer.
getMessage()	Returns the message.
getMessageId()	Returns the message ID.
getPayload()	Returns the payload.
getQueueName()	Returns the name of the queue.

13.21.1 getConsumerName()

Gets the name of the consumer for which the message has been enqueued. In a single consumer queue, this is a empty string.

Copyright © 1999, 2025, Oracle and/or its affiliates.

```
string getConsumerName() const;
```

13.21.2 getMessage()

Gets the message which has been enqueued into the non-persistent queue.

```
Message getMessage() const;
```

13.21.3 getMessageId()

Gets the id of the message which has been enqueued.

```
Bytes getMessageId() const;
```

13.21.4 getPayload()

Gets the payload in case of a notification from `NS_ANONYMOUS` namespace.

```
Bytes getPayload() const;
```

13.21.5 getQueueName()

Gets the name of the queue on which the enqueue has happened

```
string getQueueName() const;
```

13.22 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*10^125
```

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.

```
/* Create a connection */
Environment *env = Environment::createEnvironment(Environment::DEFAULT);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
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 nu1 (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

Method	Summary
Number()	`Number` class constructor.
abs()	Returns the absolute value of the number.
arcCos()	Returns the arcCosine of the number.
arcSin()	Returns the arcSine of the number.
arcTan()	Returns the arcTangent of the number.
arcTan2()	Returns the arcTangent2 of the input number `y` and this number `x`.
ceil()	Returns the smallest integral value not less than the value of the number.
cos()	Returns the cosine of the number.
exp()	Returns the natural exponent of the number.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-30 (Cont.) Summary of Number Methods

Method	Summary
floor()	Returns the largest integral value not greater than the value of the number.
fromBytes()	Returns a Number derived from a `Bytes` object.
fromText()	Returns a Number from a given number string, format string and NLS parameters specified.
hypCos()	Returns the hyperbolic cosine of the number.
hypSin()	Returns the hyperbolic sine of the number.
hypTan()	Returns the hyperbolic tangent of the number.
intPower()	Returns the number raised to the integer value specified.
isNull()	Checks if `Number` is `NULL`.
ln()	Returns the natural logarithm of the number.
log()	Returns the logarithm of the number to the base value specified.
operator++()	Increments the `number`.
operator--()	Decrements the `number`.
operator*()	Returns the product of two `Numbers`.
operator/()	Returns the quotient of two `Numbers`.
operator%()	Returns the modulo of two `Numbers`.
operator+()	Returns the sum of two `Numbers`.
operator-()	Returns the negated value of `Number`.
operator-()	Returns the difference between two `Numbers`.
operator<()	Checks if a number is less than an other number.
operator<=()	Checks if a number is less than or equal to an other number.
operator>()	Checks if a number is greater than an other number.
operator>=()	Checks if a number is greater than or equal to an other number.
operator=()	Assigns one number to another.
operator==()	Checks if two numbers are equal.
operator!=()	Checks if two numbers are not equal.
operator*=()	Multiplication assignment.
operator/=()	Division assignment.
operator%=()	Modulo assignment.
operator+=()	Addition assignment.
operator-=()	Subtraction assignment.
operator char()	Returns `Number` converted to native char.
operator signed char()	Returns `Number` converted to native signed char.
operator double()	Returns `Number` converted to a native double.
operator float()	Returns `Number` converted to a native float.
operator int()	Returns `Number` converted to native integer.
operator long()	Returns `Number` converted to native long.
operator long double()	Returns `Number` converted to a native long double.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-30 (Cont.) Summary of Number Methods

Method	Summary
operator short()	Returns `Number` converted to native short integer.
operator unsigned char()	Returns `Number` converted to an unsigned native char.
operator unsigned int()	Returns `Number` converted to an unsigned native integer.
operator unsigned long()	Returns `Number` converted to an unsigned native long.
operator unsigned short()	Returns `Number` converted to an unsigned native short integer.
power()	Returns `Number` raised to the power of another number specified.
prec()	Returns `Number` rounded to digits of precision specified.
round()	Returns `Number` rounded to decimal place specified. Negative values are allowed.
setNull()	Sets `Number` to `NULL`.
shift()	Returns a `Number` that is equivalent to the passed value `* 10^n`, where `n` may be positive or negative.
sign()	Returns the sign of the value of the passed value: -1 for the passed value < 0, 0 for the passed value == 0, and 1 for the passed value > 0.
sin()	Returns sine of the number.
squareroot()	Returns the square root of the number.
tan()	Returns tangent of the number.
toBytes()	Returns a `Bytes` object representing the `Number`.
toText()	Returns the number as a string formatted based on the format and NLS parameters.
trunc()	Returns a `Number` with the value truncated at `n` decimal place(s). Negative values are allowed.

13.22.1 Number()

`Number` class constructor.

Syntax	Description
`Number();`	Default constructor.
`Number(` `  const Number &srcNum);`	Creates a copy of a `Number`.
`Number(` `  long double &val);`	Translates a native long double into a `Number`. The `Number` is created using the precision of the platform-specific constant `LDBL_DIG`.
`Number(` `  double val);`	Translates a native double into a `Number`. The `Number` is created using the precision of the platform-specific constant `DBL_DIG`.
`Number(` `  float val);`	Translates a native float into a `Number`. The `Number` is created using the precision of the platform-specific constant `FLT_DIG`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`Number( long val);`	Translates a native long into a `Number`.
`Number( int val);`	Translates a native int into a `Number`.
`Number( shot val);`	Translates a native short into a `Number`.
`Number( char val);`	Translates a native char into a `Number`.
`Number( signed char val);`	Translates a native signed char into a `Number`.
`Number( unsigned long val);`	Translates an native unsigned long into a `Number`.
`Number( unsigned int val);`	Translates a native unsigned int into a `Number`.
`Number( unsigned short val);`	Translates a native unsigned short into a `Number`.
`Number( unsigned char val);`	Translates the unsigned character array into a `Number`.

Parameter	Description
`srcNum`	The source `Number` copied into the new `Number` object.
`val`	The value assigned to the `Number` object.

13.22.2 abs()

This method returns the absolute value of the `Number` object.

Syntax

```
const Number abs() const;
```

13.22.3 arcCos()

This method returns the arccosine of the `Number` object.

Syntax

```
const Number arcCos() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.22.4 arcSin()

This method returns the arcsine of the `Number` object.

Syntax

```
const Number arcSin() const;
```

13.22.5 arcTan()

This method returns the arctangent of the `Number` object.

Syntax

```
const Number arcTan() const;
```

13.22.6 arcTan2()

This method returns the arctangent of the `Number` object with the parameter specified. It returns `atan2 (val, x)` where `val` is the parameter specified and `x` is the current number object.

Syntax

```
const Number arcTan2(
   const Number &val) const;
```

Parameter	Description
`val`	Number parameter `val` to the arcTangent function `atan2(val,x)`.

13.22.7 ceil()

This method returns the smallest integer that is greater than or equal to the `Number` object.

Syntax

```
const Number ceil() const;
```

13.22.8 cos()

This method returns the cosine of the `Number` object.

Syntax

```
const Number cos() const;
```

13.22.9 exp()

This method returns the natural exponential of the `Number` object.

Syntax

```
const Number exp() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.22.10 floor()

This method returns the largest integer that is less than or equal to the `Number` object.

Syntax

```
const Number floor() const;
```

13.22.11 fromBytes()

This method returns a `Number` object represented by the byte string specified.

Syntax

```
void fromBytes(
   const Bytes &str);
```

Parameter	Description
`str`	A byte string.

13.22.12 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`

Syntax	Description

`void fromText(`
`  const Environment *envp,`
`  const string &number,`
`  const string &fmt,`
`  const string &nlsParam = "");`
	Sets `Number` object to value represented by a `string`.

`void fromText(`
`  const Environment *envp,`
`  const UString &number,`
`  const UString &fmt,`
`  const UString &nlsParam);`
	Sets `Number` object to value represented by a `UString`.

Parameter	Description
`envp`	The OCCI environment.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
number	The number string to be converted to a Number object.
fmt	The format string.
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.

13.22.13 hypCos()

This method returns the hypercosine of the Number object.

Syntax

```
const Number hypCos() const;
```

13.22.14 hypSin()

This method returns the hypersine of the Number object.

Syntax

```
const Number hypSin() const;
```

13.22.15 hypTan()

This method returns the hypertangent of the Number object.

Syntax

```
const Number hypTan() const;
```

13.22.16 intPower()

This method returns a Number whose value is the number object raised to the power of the value specified.

Syntax

```
const Number intPower(
   int val) const;
```

Parameter	Description
val	Power to which the number is raised.

13.22.17 isNull()

This method tests whether the Number object is NULL. If the Number object is NULL, then TRUE is returned; otherwise, FALSE is returned.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool isNull() const;
```

13.22.18 ln()

This method returns the natural logarithm of the `Number` object.

Syntax

```
const Number ln() const;
```

13.22.19 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;
```

Parameter	Description
`val`	The base to be used in the logarithm calculation.

13.22.20 operator++()

Unary `operator++()`. This is a postfix operator.

Syntax	Description
`Number& operator++();`	This method returns the `Number` object incremented by 1.
`const Number operator++(` `  int incr);`	This method returns the `Number` object incremented by the integer specified.

Parameter	Description
`incr`	The number by which the `Number` object is incremented.

13.22.21 operator--()

Unary `operator--()`. This is a prefix operator.

Syntax	Description
`Number& operator--();`	This method returns the `Number` object decremented by 1.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`const Number operator--(` `  int decr);`	This method returns the `Number` object decremented by the integer specified.

Parameter	Description
`decr`	The number by which the Number object is decremented.

13.22.22 operator*()

This method returns the product of the parameters specified.

Syntax

```
Number operator*(
   const Number &first,

   const Number &second);
```

Parameter	Description
`first`	First multiplicand.
`second`	Second multiplicand.

13.22.23 operator/()

This method returns the quotient of the parameters specified.

Syntax

```
Number operator/(
   const Number &dividend,
   const Number &divisor);
```

Parameter	Description
`dividend`	The number to be divided.
`divisor`	The number by which to divide.

13.22.24 operator%()

This method returns the remainder of the division of the parameters specified.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
Number operator%(
   const Number &dividend,
   const Number &divider);
```

Parameter	Description
`dividend`	The number to be divided.
`divizor`	The number by which to divide.

13.22.25 operator+()

This method returns the sum of the parameters specified.

Syntax

```
Number operator+(
   const Number &first,
   const Number &second);
```

Parameter	Description
`first`	First number to be added.
`second`	Second number to be added.

13.22.26 operator-()

Unary `operator-()`. This method returns the negated value of the `Number` object.

Syntax

```
const Number operator-();
```

13.22.27 operator-()

This method returns the difference between the parameters specified.

Syntax

```
Number operator-(
   const Number &subtrahend,
   const Number &subtractor);
```

Parameter	Description
`subtrahend`	The number to be reduced.
`subtractor`	The number to be subtracted.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.22.28 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);
```

Parameter	Description
first	First number to be compared.
second	Second number to be compared.

13.22.29 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);
```

Parameter	Description
first	First number to be compared.
second	Second number to be compared.

13.22.30 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

```
bool operator>(
   const Number &first,
   const Number &second);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
`first`	First number to be compared.
`second`	Second number to be compared.

13.22.31 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

```
bool operator>=(
   const Number &first,
   const Number &second);
```

Parameter	Description
`first`	First number to be compared.
`second`	Second number to be compared.

13.22.32 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

```
bool operator==(
   const Number &first,
   const Number &second);
```

Parameter	Description
`first`	First number to be compared.
`second`	Second number to be compared.

13.22.33 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool operator!=(
   const Number &first,
   const Number &second);
```

Parameter	Description
first	First number to be compared.
second	Second number to be compared.

13.22.34 operator=()

This method assigns the value of the parameter specified to the Number object.

Syntax

```
Number& operator=(
   const Number &num);
```

Parameter	Description
num	A parameter of type Number.

13.22.35 operator*=()

This method multiplies the Number object by the parameter specified, and assigns the product to the Number object.

Syntax

```
Number& operator*=(
   const Number &num);
```

Parameter	Description
num	A parameter of type Number.

13.22.36 operator/=()

This method divides the Number object by the parameter specified, and assigns the quotient to the Number object.

Syntax

```
Number& operator/=(
   const Number &num);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
num	A parameter of type Number.

13.22.37 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);
```

Parameter	Description
num	A parameter of type Number.

13.22.38 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);
```

Parameter	Description
num	A parameter of type Number.

13.22.39 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);
```

Parameter	Description
num	A parameter of type Number.

13.22.40 operator char()

This method returns the value of the Number object converted to a native char.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
operator char() const;
```

13.22.41 operator signed char()

This method returns the value of the Number object converted to a native `signed char`.

Syntax

```
operator signed char() const;
```

13.22.42 operator double()

This method returns the value of the `Number` object converted to a native `double`.

Syntax

```
operator double() const;
```

13.22.43 operator float()

This method returns the value of the `Number` object converted to a native `float`.

Syntax

```
operator float() const;
```

13.22.44 operator int()

This method returns the value of the `Number` object converted to a native `int`.

Syntax

```
operator int() const;
```

13.22.45 operator long()

This method returns the value of the `Number` object converted to a native `long`.

Syntax

```
operator long() const;
```

13.22.46 operator long double()

This method returns the value of the `Number` object converted to a native `long double`.

Syntax

```
operator long double() const;
```

13.22.47 operator short()

This method returns the value of the `Number` object converted to a native `short` integer.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
operator short() const;
```

13.22.48 operator unsigned char()

This method returns the value of the `Number` object converted to a native `unsigned char`.

Syntax

```
operator unsigned char() const;
```

13.22.49 operator unsigned int()

This method returns the value of the `Number` object converted to a native `unsigned int`.

Syntax

```
operator unsigned int() const;
```

13.22.50 operator unsigned long()

This method returns the value of the `Number` object converted to a native `unsigned long`.

Syntax

```
operator unsigned long() const;
```

13.22.51 operator unsigned short()

This method returns the value of the `Number` object converted to a native `unsigned short` integer.

Syntax

```
operator unsigned short() const;
```

13.22.52 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;
```

Parameter	Description
`val`	The power to which the number has to be raised.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.22.53 prec()

This method returns the value of the `Number` object rounded to the digits of precision provided by the parameter specified.

Syntax

```
const Number prec(
   int digits) const;
```

Parameter	Description
`digits`	The number of digits of precision.

13.22.54 round()

This method returns the value of the `Number` object rounded to the decimal place provided by the parameter specified.

Syntax

```
const Number round(
   int decPlace) const;
```

Parameter	Description
`decPlace`	The number of digits to the right of the decimal point.

13.22.55 setNull()

This method sets the value of the `Number` object to `NULL`.

Syntax

```
void setNull();
```

13.22.56 shift()

This method returns the `Number` object multiplied by 10 to the power provided by the parameter specified.

Syntax

```
const Number shift(
   int val) const;
```

Parameter	Description
`val`	An integer value.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.22.57 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;
```

13.22.58 sin()

This method returns the sin of the `Number` object.

Syntax

```
const Number sin() const;
```

13.22.59 squareroot()

This method returns the square root of the `Number` object.

Syntax

```
const Number squareroot() const;
```

13.22.60 tan()

This method returns the tangent of the `Number` object.

Syntax

```
const Number tan() const;
```

13.22.61 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 `0`th byte is the exponent byte.

Syntax

```
Bytes toBytes() const;
```

13.22.62 toText()

Convert the `Number` object to a formatted `string` or `UString` based on the parameters specified.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle Database SQL Language Reference*for information on `TO_NUMBER`

Syntax	Description

`string toText(`
`  const Environment *envp,`
`  const string &fmt,`
`  const string &nlsParam = "") const;`
	Convert the `Number` object to a formatted `string` based on the parameters specified.

`UString toText(`
`  const Environment *envp,`
`  const UString &fmt,`
`  const UString &nlsParam) const;`
	Convert the `Number` object to a `UString` based on the parameters specified.

Parameter	Description
`envp`	The OCCI environment.
`fmt`	The format string.
`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`.

13.22.63 trunc()

This method returns the `Number` object truncated at the number of decimal places provided by the parameter specified.

Syntax

```
const Number trunc(
   int decPlace) const;
```

Parameter	Description
`decPlace`	The number of places to the right of the decimal place at which the value is to be truncated.

13.23 PObject Class

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

Copyright © 1999, 2025, Oracle and/or its affiliates.

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:

```
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

Attribute	Options
`LockOption`	• `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 `SELECT FOR UPDATE` • `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 `SELECT FOR UPDATE WITH NOWAIT`
`UnpinOption`	• `OCCI_PINCOUNT_RESET` resets the object's pin count to `0` • `OCCI_PINCOUNT_DECR` decrements the object's pin count by `1`

Table 13-32 Summary of PObject Methods

Method	Summary
PObject()	`PObject` class constructor.
flush()	Flushes a modified persistent object to the database server.
getConnection()	Returns the connection from which the `PObject` object was instantiated.
getRef()	Returns a reference to a given persistent object.
getSQLTypeName()	Returns the Oracle AI Database typename for this class.
isLocked()	Tests whether the persistent object is locked.
isNull()	Tests whether the object is `NULL`.
lock()	Lock a persistent object on the database server. The default mode is to wait for the lock if not available.
markDelete()	Marks a persistent object as deleted.
markModified()	Marks a persistent object as modified or dirty.
operator=()	Assigns one `PObject` to another.
operator delete()	Remove the persistent object from the application cache only.
operator new()	Creates a new persistent / transient instance.
pin()	Pins an object.
setNull()	Sets the object value to `NULL`.
unmark()	Unmarks an object as dirty.
unpin()	Unpins an object. In the default mode, the pin count of the object is decremented by one.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.23.1 PObject()

`PObject` class constructor.

Syntax	Description
`PObject();`	Creates a `NULL PObject`.
`PObject(` `  const PObject &obj);`	Creates a copy of `PObject`.

Parameter	Description
`obj`	The source object.

13.23.2 flush()

This method flushes a modified persistent object to the database server.

Syntax

```
void flush();
```

13.23.3 getConnection()

Returns the connection from which the persistent object was instantiated.

Syntax

```
const Connection *getConnection() const;
```

13.23.4 getRef()

This method returns a reference to the persistent object.

Syntax

```
RefAny getRef() const;
```

13.23.5 getSQLTypeName()

Returns the Oracle AI Database typename for this class.

Syntax

```
string getSQLTypeName() const;
```

13.23.6 isLocked()

This method test whether the persistent object is locked. If the persistent object is locked, then `TRUE` is returned; otherwise, `FALSE` is returned.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool isLocked() const;
```

13.23.7 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

```
bool isNull() const;
```

13.23.8 lock()

Locks a persistent object on the database server.

Syntax

```
void lock(
   PObject::LockOption lock_option);
```

Parameter	Description
`lock_option`	Locking options; see Table 13-31.

13.23.9 markDelete()

This method marks a persistent object as deleted.

Syntax

```
void markDelete();
```

13.23.10 markModified()

This method marks a persistent object as modified or dirty.

Syntax

```
void mark_Modified();
```

13.23.11 operator=()

This method assigns the value of a persistent object this `PObject` object. The nature (transient or persistent) of the object is maintained. `NULL` information is copied from the source instance.

Syntax

```
PObject& operator=(
   const PObject& obj);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
obj	The object from which the assigned value is obtained.

13.23.12 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

```
void operator delete(
   void *obj,
   size_t size);
```

Parameter	Description
obj	The pointer to object to be deleted
size	(Optional) Size is implicitly obtained from the object

13.23.13 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	Description
`void *operator new(` `  size_t size);`	Creates a default new object, with a size specification only

`void *operator new(`
`  size_t size,`
`  const Connection *conn,`
`  const string& tableName,`
`  const char *typeName);`
	Used for creating transient objects when client side characterset is multibyte.

`void *operator new(`
`  size_t size,`
`  const Connection *conn,`
`  const string& tableName,`
`  const string& typeName,`
`  const string& schTableName="",`
`  const string& schTypeName="");`
	Used for creating persistent objects when client side characterset is multibyte.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void *operator new(`
`  size_t size,`
`  const Connection *conn,`
`  const UString& tableName,`
`  const UString& typeName,`
`  const UString& schTableName="",`
`  const UString& schTypeName="");`
	Used for creating persistent objects when client side characterset is unicode (UTF16).

Parameter	Description
`size`	size of the object
`conn`	The connection to the database in which the persistent object is to be created.
`tableName`	The name of the table in the database server.
`typeName`	The SQL type name corresponding to this C++ class. The format is *<schemaname>.<typename>*.
`schTableName`	The schema table name.
`schTypeName`	The schema type name.

13.23.14 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

```
void pin();
```

13.23.15 setNull()

This method sets the object value to `NULL`.

Syntax

```
void setNull();
```

13.23.16 unmark()

This method unmarks a persistent object as modified or deleted.

Syntax

```
void unmark();
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.23.17 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 the references (`Refs`) pointing to this object. The cache sets the object eligible to be freed, if necessary, reclaiming memory.

Syntax

```
void unpin(
   UnpinOption mode=OCCI_PINCOUNT_DECR);
```

Parameter	Description
`mode`	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`.

13.24 Producer Class

The `Producer` enqueues `Messages` into a queue and defines the enqueue options.

Table 13-33 Enumerated Values Used by Producer Class

Attribute	Options
`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

Method	Summary
Producer()	`Producer` class constructor.
getQueueName()	Retrieves the name of a queue on which the `Messages` is enqueued.
getRelativeMessageId()	Retrieves the `Message` id that is referenced in a sequence deviation operation.
getSequenceDeviation()	Retrieves information regarding whether the `Message` should be dequeued ahead of other `Messages` in the queue.
getTransformation()	Retrieves the transformation applied before a `Message` is enqueued.
getVisibility()	Retrieves the transactional behavior of the enqueue request.
isNull()	Tests whether the `Producer` is `NULL`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-34 (Cont.) Summary of Producer Methods

Method	Summary
send()	Enqueues and sends a `Message`.
operator=()	Assignment operator for `Producer`.
setNull()	Frees memory if the scope of the `Producer` extends beyond the `Connection` on which it was created.
setQueueName()	Specifies the name of a queue on which the `Messages` is enqueued.
setRelativeMessageId()	Specifies the `Message` id to be referenced in the sequence deviation operation.
setSequenceDeviation()	Specifies whether `Message` should be dequeued before other `Messages` in the queue.
setTransformation()	Specifies transformation applied before enqueuing a `Message`.
setVisibility()	Specifies transaction behavior of the enqueue request.

13.24.1 Producer()

`Producer` object constructor.

Syntax	Description
`Producer(` `  const Connection *conn);`	Creates a `Producer` object with the specified `Connection`.

`Producer(`
`  const Connection *conn,`
`  const string& queue);`
	Creates a `Producer` object with the specified `Connection` and queue name.

Parameter	Description
`conn`	The connection of the new `Producer` object.
`queue`	The queue that is used by the new `Producer` object.

13.24.2 getQueueName()

Retrieves the name of a queue on which the `Messages` are enqueued.

Syntax

```
string getQueueName() cosnt;
```

13.24.3 getRelativeMessageId()

Retrieves the `Message` id that is referenced in a sequence deviation operation. Used only if a sequence deviation is specified; ignored otherwise.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
Bytes getRelativeMessageId() const;
```

13.24.4 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.

Syntax

```
EnqueueSequence getSequenceDeviation() const;
```

13.24.5 getTransformation()

Retrieves the transformation applied before a `Message` is enqueued.

Syntax

```
string getTransformation() const;
```

13.24.6 getVisibility()

Retrieves the transactional behavior of the enqueue request. `Visibility` is defined in Table 13-33.

Syntax

```
Visibility getVisibility() const;
```

13.24.7 isNull()

Tests whether the `Producer` is `NULL`. If the `Producer` is `NULL`, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

13.24.8 operator=()

The assignment operator for `Producer`.

Syntax

```
void operator=(
   const Producer& prod);
```

Parameter	Description
`prod`	The original `Producer`

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.24.9 send()

Enqueues and sends a `Message`.

Syntax	Description
`Bytes send( Message& msg);`	Used when `queueName` has been previously set by the setQueueName() method.
`Bytes send( Message& msg, string& queue);`	Enqueue the `Message` to the specified `queueName`.

Parameter	Description
`msg`	The `Message` that is enqueued.
`queue`	The name of a valid queue in the database.

13.24.10 setNull()

Frees memory associated with the `Producer`. Unless working in inner scope, this call should be made before terminating the `Connection`.

Syntax

```
void setNull();
```

13.24.11 setQueueName()

Specifies the name of a queue on which the `Messages` are enqueued. Typically used when enqueuing multiple messages to the same queue.

Syntax

```
void setQueueName(
   const string& queue);
```

Parameter	Description
`queue`	The name of a valid queue in the database, to which the `Messages` are enqueued.

13.24.12 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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setRelativeMessageId(
   const Bytes& msgid);
```

Parameter	Description
`msgid`	The id of the relative `Message`.

13.24.13 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

```
void setSequenceDeviation(
   EnqueueSequence option);
```

Parameter	Description
`option`	The enqueue sequence being set, defined in Table 13-33.

13.24.14 setTransformation()

Specifies transformation function applied before enqueuing the `Message`.

Syntax

```
void setTransformation(
   string &fName);
```

Parameter	Description
`fName`	SQL transformation function.

13.24.15 setVisibility()

Specifies transaction behavior of the enqueue request. Can be set for each enqueuing of a `Message`.

Syntax

```
void setVisibility(
   Visibility option);
```

Parameter	Description
`option`	Visibility option being set, defined in Table 13-33.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.25 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

Attribute	Options
`LockOptions`	• `OCCI_LOCK_NONE` clears the lock setting on the `Ref` object. • `OCCI_LOCK_X` indicates that the object should be locked, and to wait for the lock to be available if the object is locked by another session. • `OCCI_LOCK_X_NOWAIT` indicates that the object should be locked, and returns an error if it is locked by another session.
`PrefetchOption`	• `OCCI_MAX_PREFETCH_DEPTH` indicates that the fetch should be done to maximum depth.

Table 13-36 Summary of Ref Methods

Method	Summary
Ref()	`Ref` object constructor.
clear()	Clears the reference.
getConnection()	Returns the connection this ref was created from.
isClear()	Checks if the Ref is cleared.
isNull()	Checks if the `Ref` is `NULL`.
markDelete()	Marks the referred object as deleted.
operator->()	Dereferences the `Ref` and pins the object if necessary.
operator*()	Dereferences the `Ref` and pins or fetches the object if necessary.
operator==()	Checks if the `Ref` and the pointer refer to the same object.
operator!=()	Checks if the `Ref` and the pointer refer to different objects.
operator=()	Assignment operator.
ptr()	Returns a pointer to a `PObject`. Dereferences the `Ref` and pins or fetches the object if necessary.
setLock()	Sets the lock option for the object referred from this.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-36 (Cont.) Summary of Ref Methods

Method	Summary
setNull()	Sets the Ref to NULL.
setPrefetch()	Sets the prefetch options for complex object retrieval.
unmarkDelete()	Unmarks for delete the object referred by this.

13.25.1 Ref()

`Ref` object constructor.

Syntax	Description
`Ref();`	Creates a `NULL Ref`.
`Ref(` `  const Ref<T> &src);`	Creates a copy of `Ref`.

Parameter	Description
`src`	The `Ref` that is being copied.

13.25.2 clear()

This method clears the `Ref` object.

Syntax

```
void clear();
```

13.25.3 getConnection()

Returns the connection from which the `Ref` object was instantiated.

Syntax

```
const Connection *getConnection() const;
```

13.25.4 isClear()

This method checks if `Ref` object is cleared.

Syntax

```
bool isClear() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.25.5 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

```
bool isNull() const;
```

13.25.6 markDelete()

This method marks the referenced object as deleted.

Syntax

```
void markDelete();
```

13.25.7 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.

Syntax	Description
`T *operator->();`	Dereferences and pins or fetches a non-`const` `Ref` object.
`const T *operator->() const;`	Dereferences and pins or fetches a `const` `Ref` object.

13.25.8 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.

Syntax	Description
`T& operator*();`	Dereferences and pins or fetches a non-`const` `Ref` object.
`const T& operator*() const;`	Dereferences and pins or fetches a `const` `Ref` object.

13.25.9 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
ref	The Ref object of the object to be compared.

13.25.10 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

```
bool operator!= (
   const Ref<T> &ref) const;
```

Parameter	Description
ref	The Ref object of the object to be compared.

13.25.11 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	Description
`Ref<T>& operator=( const Ref<T> &src);`	Assigns a Ref to a Ref.
`Ref<T>& operator=( const T *)obj;`	Assigns a Ref to an object.

Parameter	Description
src	The source Ref object to be assigned.
obj	The source object pointer whose Ref object is to be assigned.

13.25.12 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	Description
`T *ptr();`	Returns a pointer of a non-const Ref object.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`const T *ptr() const;`	Returns a pointer of a `const Ref` object.

13.25.13 setLock()

This method specifies how the object should be locked when dereferenced.

Syntax

```
void setLock(lockOptions);
```

Argument	Description
`lockOptions`	The lock options as defined by `LockOptions` in Table 13-35 .

13.25.14 setNull()

This method sets the `Ref` object to `NULL`.

Syntax

```
void setNull();
```

13.25.15 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	Description

`void setPrefetch(`
`  const string &typeName,`
`  unsigned int depth);`
	Sets the prefetching options for complex object retrieval, using type name and depth.

`void setPrefetch(` `  unsigned int depth);`	Sets the prefetching options for complex object retrieval, using depth only.

`void setPrefetch(`
`  const string &schName,`
`  const string &typeName,`
`  unsigned int depth);`
	Sets the prefetching options for complex object retrieval, using schema, type name, and depth.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void setPrefetch(`
`  const UString &schName,`
`  const UString &typeName,`
`  unsigned int depth);`
	Sets the prefetching options for complex object retrieval, using schema, type name, and depth, and `UString` support.

Parameter	Description
`typeName`	Type of the Ref attribute to be prefetched.
`schName`	Schema name of the Ref attribute to be prefetched.
`depth`	Depth level to which the links should be followed; can use `PrefetchOption` as defined in Table 13-35 .

13.25.16 unmarkDelete()

This method unmarks the referred object as dirty and available for deletion.

Syntax

```
void unmarkDelete();
```

13.26 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

Method	Summary
RefAny()	Constructor for `RefAny` class.
clear()	Clears the reference.
getConnection()	Returns the connection this ref was created from.
isNull()	Checks if the `RefAny` object is `NULL`.
markDelete()	Marks the object as deleted.
operator=()	Assignment operator for `RefAny`.
operator==()	Checks if this `RefAny` object equals a specified `RefAny`.
operator!=()	Checks if not equal.
unmarkDelete()	Unmarks the object as deleted.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.26.1 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();`	Creates a `NULL RefAny`.

`RefAny(`
`  const Connection *sessptr,`
`  const OCIRef *ref);`
	Creates a `RefAny` from a session pointer and a reference.

`RefAny(` `  const RefAny& src);`	Creates a `RefAny` as a copy of another `RefAny` object.

Parameter	Description
`sessptr`	Session pointer
`ref`	A reference
`src`	The source `RefAny` object to be assigned

13.26.2 clear()

This method clears the reference.

Syntax

```
void clear();
```

13.26.3 getConnection()

Returns the connection from which this reference was instantiated.

Syntax

```
const Connection* getConnection() const;
```

13.26.4 isNull()

Returns `TRUE` if the object pointed to by this ref is `NULL` else `FALSE`.

Syntax

```
bool isNull() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.26.5 markDelete()

This method marks the referred object as deleted.

Syntax

```
void markDelete();
```

13.26.6 operator=()

Assignment operator for `RefAny`.

Syntax

```
RefAny& operator=(
   const RefAny& src);
```

Parameter	Description
`src`	The source RefAny object to be assigned.

13.26.7 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

```
bool operator== (
   const RefAny &refAnyR) const;
```

Parameter	Description
`refAnyR`	`RefAny` object to which the comparison is made.

13.26.8 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

```
bool operator!= (
   const RefAny &refAnyR) const;
```

Parameter	Description
`refAnyR`	`RefAny` object to which the comparison is made.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.26.9 unmarkDelete()

This method unmarks the referred object as dirty.

Syntax

```
void unmarkDelete();
```

13.27 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 `get`*`xxx`*`()` 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 `get`*`xxx`*`()` methods, OCCI 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::get`*`xxx`*`()` 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

Attribute	Options
`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

Method	Description
cancel()	Cancels the `ResultSet`.
closeStream()	Closes the specified `Stream`.
getBDouble()	Returns the value of a column in the current row as a `BDouble`.
getBfile()	Returns the value of a column in the current row as a `Bfile`.
getBFloat()	Returns the value of a column in the current row as a `BFloat`.
getBlob()	Returns the value of a column in the current row as a `Blob` object.
getBytes()	Returns the value of a column in the current row as a `Bytes` array.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-39 (Cont.) Summary of ResultSet Methods

Method	Description
getCharSet()	Returns the character set in which data would be fetched.
getCharSetUString()	Returns the character set in which data would be fetched as a `UString`.
getClob()	Returns the value of a column in the current row as a `Clob` object.
getColumnListMetaData()	Returns the describe information of the result set columns as a `MetaData` object.
getCurrentStreamColumn()	Returns the column index of the current readable `Stream`.
getCurrentStreamRow()	Returns the current row of the `ResultSet` being processed.
getCursor()	Returns the nested cursor as a `ResultSet`.
getDate()	Returns the value of a column in the current row as a `Date` object.
getDatabaseNCHARParam()	Returns whether data is in NCHAR character set or not.
getDouble()	Returns the value of a column in the current row as a C++ double.
getFloat()	Returns the value of a column in the current row as a C++ float.
getInt()	Returns the value of a column in the current row as a C++ int.
getIntervalDS()	Returns the value of a column in the current row as a `IntervalDS`.
getIntervalYM()	Returns the value of a column in the current row as a `IntervalYM`.
getMaxColumnSize()	Returns the value set by `setMaxColumnSize()`.
getNumArrayRows()	Returns the actual number of rows fetched in the last array fetch.
getNumber()	Returns the value of a column in the current row as a `Number` object.
getObject()	Returns the value of a column in the current row as a `PObject`.
getRef()	Returns the value of a column in the current row as a `Ref`.
getRowid()	Returns the current `ROWID` for a `SELECT FOR UPDATE` statement.
getRowPosition()	Returns the row id of the current row position.
getStatement()	Returns the `Statement` of the `ResultSet`.
getStream()getStream()	Returns the value of a column in the current row as a `Stream`.
getString()	Returns the value of a column in the current row as a string.
getTimestamp()	Returns the value of a column in the current row as a `Timestamp` object.
getUInt()	Returns the value of a column in the current row as a C++ unsigned int

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-39 (Cont.) Summary of ResultSet Methods

Method	Description
getUString()	Returns the value of a column in the current row as a `UString`.
getVector()	Returns the specified collection parameter as a vector.
getVectorOfRefs()	Returns the column in the current position as a vector of `Refs`.
isNull()	Checks whether the value is `NULL`.
isTruncated()	Checks whether truncation has occurred.
next()	Makes the next row the current row in a `ResultSet`.
preTruncationLength()	Returns the actual length of the parameter before truncation.
setBinaryStreamMode()	Specifies that a column is to be returned as a binary stream.
setCharacterStreamMode()	Specifies that a column is to be returned as a character stream.
setCharSet()	Specifies the character set in which the data is to be returned.
setCharSetUString()	Specifies the character set in which the data is to be returned.
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.
setDataBuffer()	Specifies the data buffer into which data is to be fetched, or the gather and scatter binds and defines made.
setErrorOnNull()	Enables Or Disables exception when `NULL` value is read.
setErrorOnTruncate()	Enables Or Disables exception when truncation occurs.
setPrefetchMemorySize()	Sets the amount of memory that is used internally by OCCI to store data fetched during each round trip to the server.
setPrefetchRowCount()	Sets the number of rows that are fetched internally by OCCI during each round trip to the server.
setMaxColumnSize()	Specifies the maximum amount of data in bytes to read from a column. It should be based on the environment's character set, in which the `env` has been created.
status()	Returns the current status of the `ResultSet`.

13.27.1 cancel()

This method cancels the result set.

Syntax

```
void cancel();
```

13.27.2 closeStream()

This method closes the stream specified by the parameter `stream`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void closeStream(
   Stream *stream);
```

Parameter	Description
`stream`	The `Stream` to be closed.

13.27.3 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.

13.27.4 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.

13.27.5 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.27.6 getBlob()

Get the value of a column in the current row as an `Blob`. Returns the column value; if the value is SQL `NULL`, the result is `NULL`.

Syntax

```
Blob getBlob(
   unsigned int colIndex);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

13.27.7 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

```
Bytes getBytes(
   unsigned int colIndex);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

13.27.8 getCharSet()

Gets the character set in which data would be fetched, as a string.

Syntax

```
string getCharSet(
   unsigned int colIndex) const;
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

13.27.9 getCharSetUString()

Gets the character set in which data would be fetched, as a string.

Syntax

```
UString getCharSetUString(
   unsigned int colIndex) const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.

13.27.10 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

```
Clob getClob(
   unsigned int colIndex);
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.

13.27.11 getColumnListMetaData()

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

```
vector<MetaData> getColumnListMetaData() const;
```

13.27.12 getCurrentStreamColumn()

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

```
unsigned int getCurrentStreamColumn() const;
```

13.27.13 getCurrentStreamRow()

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

```
unsigned int getCurrentStreamRow() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.27.14 getCursor()

Get the nested cursor as an `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 CURSOR`s 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);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

13.27.15 getDatabaseNCHARParam()

Returns whether data is in `NCHAR` character set or not.

Syntax

```
bool getDatabaseNCHARParam(
   unsigned int paramIndex) const;
```

Parameter	Description
`paramIndex`	Parameter index, first parameter is 1, second is 2, and so on.

13.27.16 getDate()

Get the value of a column in the current row as a `Date` object. Returns the column value; if the value is SQL `NULL`, the result is `NULL`.

Syntax

```
Date getDate(
   unsigned int colIndex);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.27.17 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);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

13.27.18 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

```
float getFloat(
   unsigned int colIndex);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

13.27.19 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

```
int getInt(
   unsigned int colIndex);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

13.27.20 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

```
IntervalDS getIntervalDS(
   unsigned int colIndex);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.

13.27.21 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.

Syntax

```
IntervalYM getIntervalYM(
   unsigned int colIndex);
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.

13.27.22 getMaxColumnSize()

Get the value set by setMaxColumnSize() .

Syntax

```
unsigned int getMaxColumnSize(
   unsigned int colIndex) const;
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.

13.27.23 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

```
unsigned int getNumArrayRows() const;
```

13.27.24 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

```
Number getNumber(
   unsigned int colIndex);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.

13.27.25 getObject()

Returns a pointer to a PObject holding the column value.

Syntax

```
PObject * getObject(
   unsigned int colIndex);
```

Parameter	Description
colIndex	Column index; first column is 1, second is 2, and so on.

13.27.26 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);
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.

13.27.27 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);
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.

13.27.28 getRowPosition()

Get the rowid of the current row position.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
Bytes getRowPosition() const;
```

13.27.29 getStatement()

This method returns the `statement` of the `ResultSet`.

Syntax

```
Statement* getStatement() const;
```

13.27.30 getStream()

This method returns the value of a column in the current row as a `Stream`.

Syntax

```
Stream * getStream(
   unsigned int colIndex);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

13.27.31 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

```
string getString(
   unsigned int colIndex);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

13.27.32 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

```
Timestamp getTimestamp(
   unsigned int colIndex);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.27.33 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

```
unsigned int getUInt(
   unsigned int colIndex);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.

13.27.34 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);
```

Parameter	Description
`colIndex`	Column index; first column is 1, second is 2, and so on.

13.27.35 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

`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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`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<Bytes> &vect);`	Used for vectors of Bytes Class.
`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.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<float> &vect);`	Used for vectors of `float` type.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<int> &vect);`	Used for vectors of `int` type.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<IntervalDS> &vect);`	Used for vectors of IntervalDS Class.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<IntervalYM> &vect);`	Used for vectors of IntervalYM Class.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`void getVector( ResultSet *rs, unsigned int colIndex, vector<Number> &vect);`	Used for vectors of Number Class.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<Ref<T>> &vect);`	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.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<RefAny> &vect);`	Used for vectors of RefAny Class.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<string> &vect);`	Used for vectors of `string` type.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<T *> &vect);`	Intended for use on platforms where partial ordering of function templates is supported.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<T> &vect);`	Intended for use on platforms where partial ordering of function templates is not supported, such as Windows NT and z/OS.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<Timestamp> &vect);`	Used for vectors of Timestamp Class.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<unsigned int> &vect);`	Used for vectors of `unsigned int` type.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<UString> &vect);`	Used for vectors of `UString` Class; globalization enabled.

Parameter	Description
`rs`	The result set
`colIndex`	Column index, first column is 1, second is 2, and so on.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
vect	The reference to the vector (OUT parameter).

13.27.36 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

```
void getVectorOfRefs(
   ResultSet *rs,
   unsigned int colIndex,
   vector< Ref<T> > &vect);
```

Parameter	Description
rs	The result set
colIndex	Column index, first column is 1, second is 2, and so on.
vect	The reference to the vector of REFs (OUT parameter).

13.27.37 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 getxxx() 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

```
bool isNull(
   unsigned int colIndex) const;
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.

13.27.38 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

```
bool isTruncated(
   unsigned int paramIndex) const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
paramIndex	Parameter index, first parameter is 1, second is 2, and so on.

13.27.39 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 get*xxx*( ) 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

```
Status next(
   unsigned int numRows =1);
```

Parameter	Description
numRows	Number of rows to fetch for array fetches.

13.27.40 preTruncationLength()

Returns the actual length of the parameter before truncation.

Syntax

```
int preTruncationLength(
   unsigned int paramIndex) const;
```

Parameter	Description
paramIndex	Parameter index, first parameter is 1, second is 2, and so on.

13.27.41 setBinaryStreamMode()

Defines that a column is to be returned as a binary stream by the getStream method.

Syntax

```
void setBinaryStreamMode(
   unsigned int colIndex,
   unsigned int size);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.
size	The amount of data to be read as a binary stream.

13.27.42 setCharacterStreamMode()

Defines that a column is to be returned as a character stream by the getStream() method.

Syntax

```
void setCharacterStreamMode(
   unsigned int colIndex,
   unsigned int size);
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.
size	The amount of data to be read as a character stream.

13.27.43 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

```
void setCharSet(
   unsigned int colIndex,
   string charSet);
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.
charSet	Desired character set, as a string.

13.27.44 setCharSetUString()

Specifies the character set value as a UString in which the data is returned.

Syntax

```
UString setCharSetUString(
   unsigned int colIndex,
   const UString &charSet);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.
charSet	Desired character set, as a string.

13.27.45 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

```
void setDatabaseNCHARParam(
   unsigned int paramIndex,
   bool isNCHAR);
```

Parameter	Description
paramIndex	Parameter index, first parameter is 1, second is 2, and so on.
isNCHAR	TRUE or FALSE.

13.27.46 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)).

For more information on the version of this method that uses 32K length parameter, see *Oracle Database SQL Language Reference*.

Syntax	Description
void setDataBuffer(unsigned int colIndex, void *buffer, Type type, sb4 size = 0, ub2 *length = NULL, sb2 *ind = NULL, ub2 *rc = NULL);	Uses ub2 length buffer. This limits VARCHAR2 and NVARCHAR2 length to 4,000 bytes, and RAW data types to 2,000 bytes.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void setDataBuffer(`
`  unsigned int colIndex,`
`  void *buffer,`
`  Type type,`
`  sb4 size = 0,`
`  ub4 *length = NULL,`
`  sb2 *ind = NULL,`
`  ub2 *rc = NULL);`
	Uses `ub4 length` buffer (32K). This increases the length of `VARCHAR2`, `NVARCHAR2` and `RAW` data types.

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.
`buffer`	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.
`type`	Type of the data that is provided (or retrieved) in the buffer.
`size`	Size of the data buffer. For array fetches, it is the size of each element of the data items.
`length`	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`.
`ind`	Pointer to an indicator variable or array (`IN`/`OUT`).
`rc`	Pointer to array of column level return codes (`OUT`).

13.27.47 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);
```

Parameter	Description
`colIndex`	Column index, first column is 1, second is 2, and so on.
`causeException`	Enable exceptions if `TRUE`. Disable if `FALSE`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.27.48 setErrorOnTruncate()

This method enables/disables exceptions when truncation occurs.

Syntax

```
void setErrorOnTruncate(
   unsigned int paramIndex,
   bool causeException);
```

Parameter	Description
`paramIndex`	Parameter index, first parameter is 1, second is 2, and so on.
`causeException`	Enable exceptions if `TRUE`. Disable if `FALSE`.

13.27.49 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

```
void setPrefetchMemorySize(
   unsigned int bytes);
```

Parameter	Description
`bytes`	Number of bytes used for storing data fetched during each server round trip.

13.27.50 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

```
void setPrefetchRowCount(
   unsigned int rowCount);
```

Parameter	Description
`rowCount`	Number of rows to fetch for each round trip to the server.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.27.51 setMaxColumnSize()

Specifies the maximum amount of data in bytes to read from a column. It should be based on the environment's character set, in which the env has been created.

Syntax

```
void setMaxColumnSize(
   unsigned int colIndex,
   unsigned int max);
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2, and so on.
max	The maximum amount of data in bytes to be read.

13.27.52 status()

Returns the current Status of the result set, as defined in Table 13-38. This method can be called repeatedly.

Syntax

```
Status status() const;
```

13.28 SQLException Class

The SQLException class provides information on generated errors, their codes and associated messages.

Table 13-40 Summary of SQLException

Method	Description
SQLException()	SQLException constructor.
getErrorCode()	Returns the database error code.
getMessage()	Returns the error message string for this exception.
getNLSMessage()	Returns the error message string for this exception (Unicode support).
getNLSUStringMessage()	Returns the error message UString for this exception (Unicode support).
getUStringMessage()	Returns the error message UString for this exception.
getXAErrorCode()	Returns the error message string for this exception.
isRecoverable()	Determines whether an application can failover and recover from an error.
setErrorCtx()	Sets the error context.
what()	Returns the error message associated with the SQLException.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.28.1 SQLException()

This is the `SQLException` constructor.

Syntax	Description
`SQLException();`	Constructs a `NULL SQLException` object.
`SQLException(` `  const SQLException &e);`	Constructs an `SQLException` object as a copy of another `SQLException` object.

Parameter	Description
`e`	The `SQLException` to be copied.

13.28.2 getErrorCode()

Gets the database error code.

Syntax

```
int getErrorCode() const;
```

13.28.3 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

```
string getMessage() const;
```

13.28.4 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

```
string getNLSMessage(
   Environment *env) const;
```

Parameter	Description
`env`	The globalization enabled environment.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.28.5 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

```
UString getNLSUStringMessage(
   Environment *env) const;
```

Parameter	Description
`env`	The globalization enabled environment.

13.28.6 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

```
UString getUStringMessage() const;
```

13.28.7 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

```
int getXAErrorCode(
   const string &dbname) const;
```

Parameter	Description
`dbname`	The database name; same as the optional `dbname` provided in the Open String and used when connecting to the Resource Manager.

13.28.8 isRecoverable()

Determines whether an application can failover and recover from an error. Returns `TRUE` if recoverable.

For example, an application may recover from `ORA-03113`, but not from `ORA-942`.

Syntax

```
bool isRecoverable();
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.28.9 setErrorCtx()

Sets the pointer to the error context.

Syntax

```
void setErrorCtx(
   void *ctx);
```

Parameter	Description
ctx	The pointer to the error context.

13.28.10 what()

Standard C++ compliant function; returns the error message associated with the SQLException.

Syntax

```
const char *what() const throw();
```

13.29 StatelessConnectionPool Class

This class represents a pool of stateless, authenticated connections to the database.

Table 13-41 Enumerated Values Used by StatelessConnectionPool Class

Attribute	Options
PoolType	• HETEROGENEOUS is the default state; connections with different authentication contexts can be created in the same pool. This pool type also supports external authentication. • HOMOGENEOUS indicates that all connections in the pool are authenticated with the username and password provided during pool creation. No proxy connections can be created. minConn and incrConn values are considered only in these HOMOGENEOUS pools. • NO_RLB turns off run-time load balancing in the connection pool. Can be used with both HETEROGENEOUS and HOMOGENEOUS pools. • USES_EXT_AUTH indicates that the connections in the pool support external authentication. Can only be used with HETEROGENEOUS pools.
BusyOption	• WAIT indicates that the thread waits and blocks until the connection becomes free. • NOWAIT throws and error. • FORCEGET indicates that a new connection is created, even when maximum number of connections is opened and all are busy.
DestroyMode	• DEFAULT indicates that if are still active busy connections in the pool, ORA24422 error is thrown • SPD_FORCE indicates that any busy connections in the pool are forcefully terminated and the pool is destroyed; the user loses memory corresponding to the number of connections forcefully terminated.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-42 Summary of StatelessConnectionPool Methods

Method	Description
getAnyTaggedConnection()	Returns a pointer to the connection object, without the restriction of a matching tag.
getAnyTaggedProxyConnection()	Returns a proxy connection from a connection pool.
getBusyConnections()	Returns the number of busy connections in the connection pool.
getBusyOption()	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
getConnection()	Returns a pointer to the `Connection` object.
getIncrConnections()	Returns the number of incremental connections in the connection pool.
getMaxConnections()	Returns the maximum number of connections in the connection pool.
getMinConnections()	Returns the minimum number of connections in the connection pool.
getOpenConnections()	Returns the number of open connections in the connection pool.
getPoolName()	Returns the name of the connection pool.
getProxyConnection()	Returns a proxy connection from a connection pool.
getTimeOut()	Returns the timeout period of a connection in the connection pool.
releaseConnection()	Releases the connection back to the pool with an optional tag.
setBusyOption()	Specifies the behavior of the stateless connection pool when: • all the connections in the pool are busy, and • the number of connections have reached maximum.
setPoolSize()	Sets the maximum, minimum, and incremental number of pooled connections for the connection pool.
setTimeOut()	Sets the timeout period of a connection in the connection pool.
terminateConnection()	Closes the connection and remove it from the pool.

13.29.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`Connection *getAnyTaggedConnection( string &tag="")=0;`	Returns a pointer to the connection object from a homogeneous stateless connection pool, without the restriction of a matching tag; `string` support.
`Connection* getAnyTaggedConnection( const UString &tag)=0;`	Returns a pointer to the connection object from a homogeneous stateless connection pool, without the restriction of a matching tag; `UString` support.
`Connection *getAnyTaggedConnection( const string &userNname, const string &password, const string &tag="")=0;`	Returns a pointer to the connection object from a heterogeneous stateless connection pool, without the restriction of a matching tag; `string` support.
`Connection* getAnyTaggedConnection( const UString &userName, const UString &Password, const UString &tag)=0 ;`	Returns a pointer to the connection object from a heterogeneous stateless connection pool, without the restriction of a matching tag; `UString` support.

Parameter	Description
`userName`	The database username
`password`	The database password.
`tag`	User-defined type of connection requested. This parameter can be ignored if a default connection is requested.

13.29.2 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`Connection *getAnyTaggedProxyConnection( const string &name, string roles[], unsigned int numRoles, const string tag="", Connection::ProxyType proxyType=Connection::PROXY_DEFAULT);`	Get a proxy connection with role specifications from a connection pool; includes support for roles and `string` support.
`Connection* getAnyTaggedProxyConnection( const UString &name, string roles[], unsigned int numRoles, const UString &tag, Connection::ProxyType proxyType = Connection::PROXY_DEFAULT);`	Get a proxy connection with role specifications from a connection pool; includes support for roles and `UString` support.
`Connection *getAnyTaggedProxyConnection( const string &name, const string tag="", Connection::ProxyType proxyType=Connection::PROXY_DEFAULT);`	Get a proxy connection with role specifications from a connection pool; `string` support.
`Connection* getAnyTaggedProxyConnection( const UString &name, const UString &tag, Connection::ProxyType proxyType = Connection::PROXY_DEFAULT);`	Get a proxy connection within role specifications from the connection pool; `UString` support.

Parameter	Description
`name`	The username.
`roles`	The roles to activate on the database server
`numRoles`	The number of roles to activate on the database server
`tag`	User defined tag associated with the connection.
`proxyType`	The type of proxy authentication to perform; `ProxyType` is defined in Table 13-11.

13.29.3 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

```
unsigned int getBusyConnections() const=0;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.29.4 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.

Syntax

```
BusyOption getBusyOption()=0;
```

13.29.5 getConnection()

Returns a pointer to the connection object of a `StatelessConnectionPool`.

This method works in an environment with enabled database resident connection pooling.

Syntax	Description
`Connection *getConnection()=0;`	Returns a connection that can be authenticated externally.
`Connection *getConnection(` `  string &tag="")=0;`	Returns an authenticated connection, with a connection pool username and password; `string` support.
`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 &connectionClass,`
`  const Connection::Purity &purity)=0;`
	Returns a pointer to the connection object from a database resident 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; `UString` support.

`Connection *getConnection(`
`  const string &userName,`
`  const string &password,`
`  const string &connectionClass,`
`  const Connection::Purity &purity)=0;`
	Returns a pointer to the connection object from a database resident connection pool; user name and password authentication; `string` support.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`Connection* getConnection(`
`  const UString &userName,`
`  const UString &password,`
`  const UString &connectionClass,`
`  const Connection::Purity &purity)=0;`
	Returns a pointer to the connection object from a database resident connection pool; user name and password authentication; `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 UString &connectionClass,`
`  const Connection::Purity &purity,`
`  const UString &tag)=0;`
	Returns a tagged connection object from a database resident connection pool; `UString` support.

Parameter	Description
`userName`	The database username.
`password`	The database password.
`tag`	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.
`connectionClass`	The class of connection used by database resident connection pool.
`purity`	The purity of connection used by database resident connection pool; either `SELF` or `NEW`.

13.29.6 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

```
unsigned int getIncrConnections() const=0;
```

13.29.7 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

```
unsigned int getMaxConnections() const=0;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.29.8 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

```
unsigned int getMinConnections() const=0;
```

13.29.9 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

```
unsigned int getOpenConnections() const=0;
```

13.29.10 getPoolName()

Returns the name of the connection pool.

Syntax

```
string getPoolName() const=0;
```

13.29.11 getProxyConnection()

Returns a proxy connection from a connection pool.

This method works in an environment with enabled database resident connection pooling.

Syntax	Description

`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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`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
`userName`	The user name.
`roles`	The roles to activate on the database server.
`numRoles`	The number of roles to activate on the database server.
`tag`	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.
`proxyType`	The type of proxy authentication to perform; `ProxyType` is defined in Table 13-11.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
connectionClass	The class of connection used by database resident connection pool.
purity	The purity of connection used by database resident connection pool; either SELF or NEW.

13.29.12 getStmtCacheSize()

Retrieves the size of the statement cache.

Syntax

```
unsigned int getStmtCacheSize() const=0;
```

13.29.13 getTimeOut()

Returns the timeout period of a connection in the connection pool.

Syntax

```
unsigned int getTimeOut() const=0;
```

13.29.14 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	Description

`void releaseConnection(`
`  Connection *connection,`
`  const string& tag="");`
	Support for string tag.

`void releaseConnection(`
`  Connection *connection,`
`  const UString &tag);`
	Support for UString tag.

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.

13.29.15 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setBusyOption(
   BusyOption busyOption)=0;
```

Parameter	Description
`busyOption`	Valid values are defined in `BusyOption` in Table 13-41.

13.29.16 setPoolSize()

Sets the maximum, minimum, and incremental number of pooled connections for the connection pool.

Syntax

```
void setPoolSize(
   unsigned int maxConn=1,
   unsigned int minConn=0,
   unsigned int incrConn=1)=0;
```

Parameter	Description
`maxConn`	The maximum number of connections in the connection pool.
`minConn`	The minimum number of connections, in homogeneous pools only.
`incrConn`	The incremental number of connections, in homogeneous pools only.

13.29.17 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

```
void setTimeOut(
   unsigned int connTimeOut=0)=0;
```

Parameter	Description
`connTimeOut`	The time out period, given in seconds.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.29.18 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

```
void setStmtCacheSize(
   unsigned int cacheSize)=0;
```

Parameter	Description
`cacheSize`	The size of the statement cache

13.29.19 terminateConnection()

Closes the connection and removes it from the pool.

This method works in an environment with enabled database resident connection pooling.

Syntax

```
void terminateConnection(
   Connection *connection)=0;
```

Parameter	Description
`connection`	The connection to be terminated

13.30 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-43 Enumerated Values used by the Statement Class

Attribute	Options
`Status`	• `NEEDS_STREAM_DATA` indicates that output `Streams` must be written for the streamed `IN` bind parameters. If there are multiple streamed parameters, call the getCurrentStreamParam() method to find out the bind parameter that needs the stream. If the statement is executed iteratively, call getCurrentIteration() to find the iteration for the stream that must to be written. • `PREPARED` indicates that the `Statement` is set to a query. • `RESULT_SET_AVAILABLE` indicates that the getResultSet() method must be called to get the result set. • `STREAM_DATA_AVAILABLE` indicates that the input `Streams` must be read for the streamed `OUT` bind parameters. If there are multiple streamed parameters, call the getCurrentStreamParam() method to find out the bind parameter that needs the stream. If the statement is executed iteratively, call getCurrentIteration() to find the iteration for the stream that must be read. • `UPREPARED` indicates that the `Statement` object is not set to a query. • `UPDATE_COUNT_AVAILABLE` indicates that the getUb8RowCount() method must be called to find out the update count.

Table 13-44 Statement Methods

Method	Description
addIteration()	Adds an iteration for execution.
closeResultSet()	Immediately releases a result set's database and OCCI resources instead of waiting for automatic release.
closeStream()	Closes the stream specified by the parameter *`stream`*.
disableCaching()	Disables statement caching.
execute()	Runs the SQL statement.
executeArrayUpdate()	Runs insert, update, and delete statements that use only the `setDataBuffer()` or stream interface for bind parameters.
executeQuery()	Runs a SQL statement that returns a single `ResultSet`.
executeUpdate()	Runs a SQL statement that does not return a `ResultSet`.
getAutoCommit()	Returns the current auto-commit state.
getBatchErrorMode()	Returns the state of the batch error mode.
getBDouble()	Returns the value of an IEEE754 `DOUBLE` as a `BDouble` object.
getBfile()	Returns the value of a `BFILE` as a `Bfile` object.
getBFloat()	Returns the value of a `IEEE754 FLOAT` as a `BFloat` object.
getBlob()	Returns the value of a `BLOB` as a `Blob` object.
getBytes()	Returns the value of a `SQL BINARY` or `VARBINARY` parameter as `Bytes`.
getCharSet()	Returns the characterset that is in effect for the specified parameter, as a `string`.
getCharSetUString()	Returns the characterset that is in effect for the specified parameter, as a `UString`.
getClob()	Returns the value of a `CLOB` as a `Clob` object.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-44 (Cont.) Statement Methods

Method	Description
getConnection()	Returns the connection from which the `Statement` object was instantiated.
getCurrentIteration()	Returns the iteration number of the current iteration that is being processed.
getCurrentStreamIteration()	Returns the current iteration for which stream data is to be read or written.
getCurrentStreamParam()	Returns the parameter index of the current output Stream that must be read or written.
getCursor()	Returns the `REF CURSOR` value of an OUT parameter as a `ResultSet`.
getDatabaseNCHARParam()	Returns whether data is in NCHAR character set.
getDate()	Returns the value of a parameter as a `Date` object
getDMLRowCounts()	Returns the row counts affected by each iteration of the current DML statement when it executes with multiple iterations.
getDouble()	Returns the value of a parameter as a C++ double.
getBFloat()	Returns the value of a parameter as an IEEE754 float.
getFloat()	Returns the value of a parameter as a C++ float.
getInt()	Returns the value of a parameter as a C++ int.
getIntervalDS()	Returns the value of a parameter as a `IntervalDS` object.
getIntervalYM()	Returns the value of a parameter as a `IntervalYM` object.
getMaxIterations()	Returns the current limit on maximum number of iterations.
getMaxParamSize()	Returns the current max parameter size limit.
getNumber()	Returns the value of a parameter as a `Number` object.
getObject()	Returns the value of a parameter as a `PObject`.
getOCIStatement()	Returns the OCI statement handle associated with the `Statement`.
getRef()	Returns the value of a REF parameter as `RefAny`
getResultSet()	Returns the current result as a `ResultSet`.
getRowCountsOption()	Determines if the DML row counts option is enabled.
getRowid()	Returns the row id parameter value as a `Bytes` object.
getSQL()	Returns the current SQL string associated with the `Statement` object.
getSQLUString()	Returns the current SQL string associated with the `Statement` object; globalization enabled.
getStream()	Returns the value of the parameter as a stream.
getString()	Returns the value of the parameter as a string.
getTimestamp()	Returns the value of the parameter as a `Timestamp` object.
getUb8RowCount()	Returns the number of rows processed.
getUInt()	Returns the value of the parameter as a C++ unsigned integer.
getUpdateCount()	Returns the number of rows processed.
getUString()	Returns the value of a `UString`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-44 (Cont.) Statement Methods

Method	Description
getVector()	Returns the specified parameter as a vector.
getVectorOfRefs()	Returns the column in the current position as a vector of REFs.
isNull()	Checks whether the parameter is NULL.
isTruncated()	Checks whether the value is truncated.
preTruncationLength()	Returns the actual length of the parameter before truncation.
registerOutParam()	Registers the type and max size of the OUT parameter.
setAutoCommit()	Specifies auto commit mode.
setBatchErrorMode()	Enables or disables the batch error processing mode.
setBDouble()	Sets a parameter to an IEEE double value.
setBfile()	Sets a parameter to a Bfile value.
setBFloat()	Sets a parameter to an IEEE float value.
setBinaryStreamMode()	Specifies that a column is to be returned as a binary stream.
setBlob()	Sets a parameter to a Blob value.
setBytes()	Sets a parameter to a Bytes array.
setCharacterStreamMode()	Specifies that a column is to be returned as a character stream.
setCharSet()	Specifies the characterset as a string.
setCharSetUString()	Specifies the character set as a UString.
setClob()	Sets a parameter to a Clob value.
setDate()	Sets a parameter to a Date value.
setDatabaseNCHARParam()	Sets to true if the data is to be in the NCHAR character set of the database; set to false to restore the default.
setDataBuffer()	Specifies a data buffer where data would be available for reading or writing.
setDataBufferArray()	Specifies an array of data buffers where data would be available for reading or writing.
setDouble()	Sets a parameter to a C++ double value.
setErrorOnNull()	Enables Or Disables exceptions for reading of NULL values.
setErrorOnTruncate()	Enables Or Disables exception when truncation occurs.
setFloat()	Sets a parameter to a C++ float value.
setInt()	Sets a parameter to a C++ int value.
setIntervalDS()	Sets a parameter to a IntervalDS value.
setIntervalYM()	Sets a parameter to a IntervalYM value.
setMaxIterations()	Sets the maximum number of invocations that area made for the DML statement.
setMaxParamSize()	Sets the maximum amount of data that can sent or returned from the parameter.
setNull()	Sets a parameter to SQL NULL.
setNumber()	Sets a parameter to a Number value.
setObject()	Sets the value of a parameter using an object.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-44 (Cont.) Statement Methods

Method	Description
setPrefetchMemorySize()	Sets the amount of memory that is used internally by OCCI to store data fetched during each round trip to the server.
setPrefetchRowCount()	Sets the number of rows that are fetched internally by OCCI during each round trip to the server.
setRef()	Sets the value of a parameter to a reference.
setRowCountsOption()	Set `flag` to TRUE to enable return of DML row counts per iteration when invoking getDMLRowCounts().
setRowid()	Sets a row id bytes array for a bind position.
setSQL()	Associates new SQL string with `Statement` object.
setSQLUString()	Associates new SQL string with `Statement` object; globalization enabled.
setString()	Sets a parameter for a specified index.
setTimestamp()	Sets a parameter to a `Timestamp` value.
setUInt()	Sets a parameter to a C++ unsigned int value.
setUString()	Sets a parameter for a specified index; globalization enabled.
setVector()	Sets a parameter to a vector of unsigned int.
setVectorOfRefs()	Sets a parameter to a vector; is necessary when the type is a collection of REFs.
status()	Returns the current status of the statement. This is useful when there is streamed data to be written.

13.30.1 addIteration()

After specifying set parameters, an iteration is added for execution.

Syntax

```
void addIteration();
```

13.30.2 closeResultSet()

Immediately releases the specified `resultSet`'s database and OCCI resources when the result set is not needed.

Syntax

```
void closeResultSet(
   ResultSet *resultSet);
```

Parameter	Description
`resultSet`	The result set to be closed; may be a result of getResultSet() , executeQuery() , or getCursor() calls on the current statement, or by a getCursor() call of another result set on the same statement.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.30.3 closeStream()

Closes the stream specified by the parameter `stream`.

Syntax

```
void closeStream(
   Stream *stream);
```

Parameter	Description
`stream`	The stream to be closed.

13.30.4 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();
```

13.30.5 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.

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.

Syntax	Description
`Status execute(` `  const string &sql="");`	Executes the SQL Statement.
`Status execute(` `  const UString &sql);`	Executes the SQL Statement; globalization enabled.

Parameter	Description
`sql`	The SQL statement to be executed. This can be `NULL` if the executeArrayUpdate() method was used to associate the sql with the statement.

13.30.6 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`

Copyright © 1999, 2025, Oracle and/or its affiliates.

parameter. The statement may have writable streams which may have to be written. The returned value `Status` is defined in Table 13-43.

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

```
Status executeArrayUpdate(
   unsigned int arrayLength);
```

Parameter	Description
`arrayLength`	The number of elements provided in each buffer of bind variables.

13.30.7 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.

Syntax	Description
`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.

Parameter	Description
`sql`	The SQL statement to be executed. This can be `NULL` if the executeArrayUpdate() method was used to associate the sql with the statement.

13.30.8 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.

If the number of rows processed as a result of this call exceeds `UB4MAXVAL`, it may throw an exception. In such scenarios, use execute() instead, followed by getUb8RowCount() to obtain the number of rows processed.

Syntax	Description
`unsigned int executeUpdate(` `  const string &sql="");`	Executes a non-query statement.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`unsigned int executeUpdate(` `  const UString &sql);`	Executes a non-query statement; globalization enabled.

Parameter	Description
`sql`	The SQL statement to be executed. This can be `NULL` if the executeArrayUpdate() method was used to associate the sql with the statement.

13.30.9 getAutoCommit()

Returns the current auto-commit state.

Syntax

```
bool getAutoCommit() const;
```

13.30.10 getBatchErrorMode()

Returns the state of the batch error mode; `TRUE` if the batch error mode is enabled, `FALSE` otherwise.

Syntax

```
bool getBatchErrorMode() const;
```

13.30.11 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

```
BDouble getBDouble(
   unsigned int paramIndex) = 0;
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.12 getBfile()

Returns the value of a `BFILE` parameter as a `Bfile` object.

Syntax

```
Bfile getBfile(
   unsigned int paramIndex);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.13 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

```
BFloat getBFloat(
   unsigned int paramIndex) = 0;
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.14 getBlob()

Returns the value of a BLOB parameter as a Blob.

Syntax

```
Blob getBlob(
   unsigned int paramIndex);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.15 getBoolean()

Gets the output values of IN/OUT and OUT parameters of the BOOLEAN type in PL/SQL procedures and functions.

Syntax

```
virtual bool getBoolean (unsigned int paramIndex) = 0;
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on. Index of BOOLEAN type.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Sample usage scenarios of `Statement::getBoolean()`

OUT BIND PL/SQL using getBoolean()

This returns an OUT bind parameter value at a given position in a statement object as a boolean value.

```
/*----------------------------------------------------------------------*//* OUT
param of boolean
        type */OCCIStatement *stmt =
        conn->createStatement("BEGIN myproc (:1); END;"); try{
        stmt->registerOutParam(1, Type::OCCIBOOL)
          stmt->execute();    bool b = stmt->getBoolean (1);          …}
/*----------------------------------------------------------------------*/
```

IN BIND PL/SQL

```
/* IN param of boolean type */
bool bVal = true;
OCCIStatement *stmt = conn->createStatement("BEGIN myproc (:1); END;");
try
{
  stmt->setBoolean(1, bVal);
  stmt->execute();
…
}
```

IN/OUT BIND PL/SQL

```
/* IN/OUT param of boolean type */
           bool bVal = true;
OCCIStatement *stmt = conn->createStatement("BEGIN myproc (:1);     END;");
try
{
  stmt->setBoolean(1, bVal);

  stmt->execute();
     bool b = stmt->getBoolean (1);          …
   }
```

Binding Boolean values in SELECT and DML statements

```
/*----------------------------------------------------------------------*/
Statement *stmt = conn->createStatement(“ INSERT INTO BoolTable
(booleanColumn) VALUES (:1)" );

try
{
  bool boolIn = true;
  stmt->setMaxIterations (1);

  stmt->setBoolean(1, boolIn);
  stmt->executeUpdate ();
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
}
/*-----------------------------------------------------------------------*/
```

Selecting BOOLEAN column values from tables and views

```
Class: ResultSet
virtual bool getBoolean(unsigned int colIndex)   = 0;
This returns the selected column value as a boolean value from the ResultSet
object
/*-----------------------------------------------------------------------*/

Statement *stmt = conn->createStatement( " SELECT * from BoolTable " );

try
{
  ResultSet rs = stmt->executeQuery();
              rs->next();
  string sname = rs->getString(1);
  bool boolOut = rs->getBoolean(2);
}
/*-----------------------------------------------------------------------*/
```

13.30.16 getBytes()

Returns the value of n SQL `BINARY` or `VARBINARY` parameter as `Bytes`; if the value is SQL `NULL`, the result is `NULL`.

Syntax

```
Bytes getBytes(
   unsigned int paramIndex);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.17 getCharSet()

Returns the characterset that is in effect for the specified parameter, as a `string`.

Syntax

```
string getCharSet(
   unsigned int paramIndex) const;
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.18 getCharSetUString()

Returns the character set that is in effect for the specified parameter, as a `UString`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
UString getCharSetUString(
   unsigned int paramIndex) const;
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.19 getClob()

Get the value of a `CLOB` parameter as a `Clob`. Returns the parameter value.

Syntax

```
Clob getClob(
   unsigned int paramIndex);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.20 getConnection()

Returns the connection from which the `Statement` object was instantiated.

Syntax

```
Connection* getConnection() const;
```

13.30.21 getCurrentIteration()

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;
```

13.30.22 getCurrentStreamIteration()

Returns the current parameter stream for which data is available.

Syntax

```
unsigned int getCurrentStreamIteration() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.30.23 getCurrentStreamParam()

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;
```

13.30.24 getCursor()

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 CURSOR`s 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);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.25 getDatabaseNCHARParam()

Returns whether data is in `NCHAR` character set or not.

Syntax

```
bool getDatabaseNCHARParam(
   unsigned int paramIndex) const;
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.26 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.27 getDMLRowCounts()

Returns the row counts affected by each iteration of the current DML statement when it executes with multiple iterations.

Use this method in conjunction with getRowCountsOption() and setRowCountsOption().

Syntax

```
vector<oraub8> getDMLRowCounts();
```

13.30.28 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);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.29 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.

Syntax

```
float getFloat(
   unsigned int paramIndex);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.30 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

```
unsigned int getInt(
   unsigned int paramIndex);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.31 getIntervalDS()

Get the value of a parameter as a IntervalDS object.

Syntax

```
IntervalDS getIntervalDS(
   unsigned int paramIndex);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.32 getIntervalYM()

Get the value of a parameter as a IntervalYM object.

Syntax

```
IntervalYM getIntervalYM(
   unsigned int paramIndex);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.33 getMaxIterations()

Gets the current limit on maximum number of iterations. Default is 1. Returns the current maximum number of iterations.

Syntax

```
unsigned int getMaxIterations() const;
```

13.30.34 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;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.35 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
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.36 getObject()

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
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.37 getOCIStatement()

Get the OCI statement handle associated with the Statement.

Syntax

```
OCIStmt * getOCIStatement() const;
```

13.30.38 getRef()

Get the value of a REF parameter as RefAny. Returns the parameter value.

Syntax

```
RefAny getRef(
   unsigned int paramIndex);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.39 getResultSet()

Returns the current result as a `ResultSet`.

Syntax

```
ResultSet * getResultSet();
```

13.30.40 getRowCountsOption()

Determines if the DML row counts option is enabled.

If `TRUE`, DML statements can be executed to return the row counts for each iteration using the method getDMLRowCounts().

Syntax

```
bool getRowCountsOption();
```

13.30.41 getRowid()

Get the `rowid` parameter value as a `Bytes`.

Syntax

```
Bytes getRowid(
   unsigned int paramIndex);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.42 getSQL()

Returns the current SQL string associated with the Statement object.

Syntax

```
string getSQL() const;
```

13.30.43 getSQLUString()

Returns the current SQL `UString` associated with the Statement object; globalization enabled.

Syntax

```
UString getSQLUString() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.30.44 getStream()

Returns the value of the parameter as a stream.

Syntax

```
Stream * getStream(
   unsigned int paramIndex);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.45 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

```
string getString(
   unsigned int paramIndex);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.46 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

```
Timestamp getTimestamp(
   unsigned int paramIndex);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.47 getUb8RowCount()

Returns the number of rows affected by the execution of a DML statement.

This method enables a return of a large number of rows than was possible before Oracle Database Release 12*c*.

Syntax

```
oraub8 getUb8RowCount();
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.30.48 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

```
unsigned int getUInt(
   unsigned int paramIndex);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.49 getUpdateCount()

Returns the number of rows affected, if DML statement is executed.

Note: This method has been deprecated. Use `getUb8RowCount()` instead.

Syntax

```
unsigned int getUpdateCount() const;
```

13.30.50 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
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.51 getVector()

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	Description

`void getVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  std::vector<UString> &vect);`
	Used for vectors of `UString Class`; globalization enabled.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`void getVector( Statement *stmt, unsigned int paramIndex, vector<BDouble> &vect);`	Used for `BDouble` vectors.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<BFile> &vect);`	Used for vectors of Bfile Class.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<BFloat> &vect);`	Used for `BFloat` vectors.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<Blob> &vect);`	Used for vectors of Blob Class.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<Bytes> &vect);`	Used for vectors of Bytes Class.
`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.`

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`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.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<Timestamp> &vect);`	Used for vectors of Timestamp Class.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<u <Ref<T> > &vect);`	Available only on platforms where partial ordering of function templates is supported.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<unsigned int> &vect);`	Used for on vectors of `unsigned int Class.`

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
stmt	The statement.
paramIndex	Parameter index.
vect	Reference to the vector (OUT parameter) into which the values should be retrieved.

13.30.52 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

```
void getVectorOfRefs(
   Statement *stmt,
   unsigned int colIndex,
   vector< Ref<T> > &vect);
```

Parameter	Description
stmt	The statement object.
colIndex	Column index; first column is 1, second is 2, and so on.
vect	The reference to the vector of REFs (OUT parameter). It is recommended to use getVectorOfRefs() instead of specialized getVector() function for Ref<T>.

13.30.53 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 get*xxx*() 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

```
bool isNull(
   unsigned int paramIndex ) const;
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

13.30.54 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool isTruncated(
   unsigned int paramIndex) const;
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.55 preTruncationLength()

Returns the actual length of the parameter before truncation.

Syntax

```
int preTruncationLength(
   unsigned int paramIndex) const;
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

13.30.56 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.

Syntax	Description

`void registerOutParam(`
`  unsigned int paramIndex,`
`  Type type,`
`  unsigned int maxSize=0,`
`  const string &sqltype="");`
	Registers the `type` of each `out` parameter of a PL/SQL stored procedure.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`void registerOutParam( unsigned int paramIndex, Type type, unsigned int maxSize, const string typName, const string &schName);`	Registers the `type` of each `out` parameter of a PL/SQL stored procedure; `string` support.
`void registerOutParam( unsigned int paramIndex, Type type, unsigned int maxSize, const UString &typName, const UString &schName);`	Registers the `type` of each `out` parameter of a PL/SQL stored procedure; globalization enabled, or `UString` support.

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`type`	SQL type code defined by *`type`*; only data types corresponding to OCCI data types such as `Date`, `Bytes`, and so on.
`maxSize`	The maximum size of the retrieved value. For data types of `OCCIBYTES` and `OCCISTRING`, `maxSize` should be greater than `0`.
`sqltype`	The name of the type in the data base (used for types which have been created with `CREATE TYPE`).
`typName`	The name of the type.
`schName`	The schema name.

13.30.57 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

```
void setAutoCommit(
   bool autoCommit);
```

Parameter	Description
`autoCommit`	`TRUE` enables auto-commit; `FALSE` disables auto-commit.

13.30.58 setBatchErrorMode()

Enables or disables the batch error processing mode.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
virtual void setBatchErrorMode(
   bool batchErrorMode);
```

Parameter	Description
batchErrorMode	TRUE enables batch error processing; FALSE disables batch error processing.

13.30.59 setBDouble()

Sets an IEEE754 double as a bind value to a `Statement` object at the position specified by `paramIndex` attribute.

Syntax

```
void setBDouble(
   unsigned int paramIndex,
   const BDouble &dval);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.
dval	The parameter value.

13.30.60 setBfile()

Sets a parameter to a `Bfile` value.

Syntax

```
void setBfile(
   unsigned int paramIndex,
   const Bfile &val);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.
val	The parameter value.

13.30.61 setBFloat()

Sets an IEEE754 float as a bind value to a `Statement` object at the position specified by the `paramIndex` attribute.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setBFloat(
   unsigned int paramIndex,
   const BFloat &fval);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`fval`	The parameter value.

13.30.62 setBinaryStreamMode()

Defines that a column is to be returned as a binary stream.

Syntax	Description

`void setBinaryStreamMode(`
`  unsigned int colIndex,`
`  unsigned int size);`
	Sets column returned to be a binary stream.

`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
`colIndex`	Column index; first column is 1, second is 2, and so on.
`size`	The amount of data to be read or returned as a binary `Stream`. This is limited to 32KB (32,768 bytes).
`inArg`	Pass `TRUE` if the bind position is a PL/SQL `IN` or `IN/OUT` argument

13.30.63 setBlob()

Sets a parameter to a `Blob` value.

Syntax

```
void setBlob(
   unsigned int paramIndex,
   const Blob &val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
`val`	The parameter value.

13.30.64 setBoolean()

Sets an input value on statement object at the given position, using the given boolean value, for PL/SQL IN and IN/OUT BOOLEAN parameters and all BOOLEAN bind values for SELECT and DML statements.

Syntax

```
virtual void setBoolean(unsigned int paramIndex, bool val) = 0;
```

Parameters	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on. This is the index of BOOLEAN type parameter of the PL/SQL procedure or function, or the bind position of the BOOLEAN type in the SELECT or DML statement executed.
`val`	`val` is the boolean value passed.

13.30.65 setBytes()

Sets a parameter to a `Bytes` array.

Syntax

```
void setBytes(
   unsigned int paramIndex,
   const Bytes &val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The parameter value.

13.30.66 setCharacterStreamMode()

Defines that a column is to be returned as a character stream.

Syntax	Description

`void setCharacterStreamMode(`
`  unsigned int colIndex,`
`  unsigned int size);`
	Sets column returned to be a character stream.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void setCharacterStreamMode(`
`  unsigned int colIndex,`
`  unsigned int size,`
`  bool inArg);`
	Sets column returned to be a character stream; used with PL/SQL `IN` or `IN/OUT` arguments in the bind position.

Parameter	Description
`colIndex`	Column index; first column is 1, second is 2, and so on.
`size`	The amount of data to be read or returned as a character `Stream`.
`inArg`	Pass `TRUE` if the bind position is a PL/SQL `IN` or `IN/OUT` argument

13.30.67 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

```
void setCharSet(
   unsigned int paramIndex,
   string &charSet);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`charSet`	Selected character set, as a string.

13.30.68 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

```
void setCharSetUString(
   unsigned int paramIndex,
   const UString& charSet);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
charSet	Selected character set, as a UString.

13.30.69 setClob()

Sets a parameter to a Clob value.

Syntax

```
void setClob(
   unsigned int paramIndex,
   const Clob &val);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.
val	The parameter value.

13.30.70 setDate()

Sets a parameter to a Date value.

Syntax

```
void setDate(
   unsigned int paramIndex,
   const Date &val);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.
val	The parameter value.

13.30.71 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 dafault.Returns returns the character set that is in effect for the specified parameter.

Syntax

```
void setDatabaseNCHARParam(
   unsigned int paramIndex,
   bool isNCHAR);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.
isNCHAR	TRUE if this parameter contains data in Database's NCHAR character set; FALSE otherwise

13.30.72 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, CHARZ 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).

For more information on the version of this method that uses 32K length parameter, see *Oracle AI Database SQL Language Reference*.

Syntax	Description
void setDataBuffer(unsigned int paramIndex, void *buffer, Type type, sb4 size, ub2 *length, sb2 *ind = NULL, ub2 *rc= NULL);	Uses ub2 length buffer. This limits VARCHAR2 and NVARCHAR2 length to 4,000 bytes, and RAW data types to 2,000 bytes.
void setDataBuffer(unsigned int paramIndex, void *buffer, Type type, sb4 size, ub4 *length, sb2 *ind = NULL, ub2 *rc= NULL);	Uses ub4 length buffer (32K). This increases the length of VARCHAR2, NVARCHAR2 and RAW data types.

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
buffer	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.
type	Type of the data that is provided (or retrieved) in the buffer.
size	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.
length	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().
ind	Indicator. For iterative and array executes, an indicator for every buffer element.
rc	Returns code; for iterative and array executes, a return code for every buffer element.

13.30.73 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.
- For more information on the version of this method that uses 32K elementLength parameter, see *Oracle AI Database SQL Language Reference*.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void setDataBufferArray(`
`  unsigned int paramIndex,`
`  void *buffer,`
`  Type type,`
`  ub4 arraySize,`
`  ub4 *arrayLength,`
`  sb4 elementSize,`
`  ub2 *elementLength,`
`  sb2 *ind = NULL,`
`  ub2 *rc = NULL);`
	Uses `ub2 elementLength` buffer. This limits `VARCHAR2` and `NVARCHAR2` length to `4,000` bytes, and `RAW` data types to `2,000` bytes.

`void setDataBufferArray(`
`  unsigned int paramIndex,`
`  void *buffer,`
`  Type type,`
`  ub4 arraySize,`
`  ub4 *arrayLength,`
`  sb4 elementSize,`
`  ub4 *elementLength,`
`  sb2 *ind = NULL,`
`  ub2 *rc = NULL);`
	Uses `ub4 elementLength` buffer (32K). This increases the length of `VARCHAR2`, `NVARCHAR2` and `RAW` data types.

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`buffer`	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.
`type`	Type of the data that is provided (or retrieved) in the buffer.
`arraySize`	Maximum number of elements in the array.
`arrayLength`	Pointer to number of current elements in the array.
`elementSize`	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`.
`elementLemgth`	Pointer to an array of lengths. `elementLength[i]` has the current length of the `i`th element of the array.
`ind`	Pointer to an array of indicators. An indicator for every buffer element.
`rcs`	Pointer to an array of return codes.

13.30.74 setDouble()

Sets a parameter to a C++ double value.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setDouble(
   unsigned int paramIndex,
   double val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The parameter value.

13.30.75 setErrorOnNull()

Enables/disables exceptions for reading of `NULL` values on paramIndex parameter of the statement. If exceptions are enabled, calling a `get`*`xxx`*`()` on `paramIndex` parameter would result in an `SQLException` if the parameter value is `NULL`. This call can also be used to disable exceptions.

Syntax

```
void setErrorOnNull(
   unsigned int paramIndex,
   bool causeException);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`causeException`	Enable exceptions if `TRUE`. Disable if `FALSE`.

13.30.76 setErrorOnTruncate()

This method enables/disables exceptions when truncation occurs.

Syntax

```
void setErrorOnTruncate(
   unsigned int paramIndex,
   bool causeException);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`causeException`	Enable exceptions if `TRUE`. Disable if `FALSE`.

13.30.77 setFloat()

Sets a parameter to a C++ float value.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setFloat(
   unsigned int paramIndex,
   float val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The parameter value.

13.30.78 setInt()

Sets a parameter to a C++ int value.

Syntax

```
void setInt(
   unsigned int paramIndex,
   int val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The parameter value.

13.30.79 setIntervalDS()

Sets a parameter to a `IntervalDS` value.

Syntax

```
void setIntervalDS(
   unsigned int paramIndex,
   const IntervalDS &val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The parameter value.

13.30.80 setIntervalYM()

Sets a parameter to a `Interval` value.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setIntervalYM(
   unsigned int paramIndex,
   const IntervalYM &val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The parameter value.

13.30.81 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);
```

Parameter	Description
`maxIterations`	Maximum number of iterations allowed on this statement.

13.30.82 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);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
maxSize	The new maximum parameter size limit; must be >0.

13.30.83 setNull()

Sets a parameter to SQL NULL. Note that you must specify the parameter's SQL type.

Syntax	Description
void setNull(unsigned int paramIndex, Type type);	Sets the value of a parameter to NULL using an object.
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, OCCIPOBJECT and OCCIVECTOR. Uses the appropriate schema and type name of the object or collection type. Support for string.
void setNull(unsigned int paramIndex, Type type, UString &typeName, UString &schemaName);	Sets the value of a parameter to NULL for object and collection types, OCCIPOBJECT and OCCIVECTOR. Uses the appropriate schema and type name of the object or collection type. Support for UString.

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.
type	SQL type.
typeName	Type name of the object or collection.
schemaName	Name of the schema where the object or collection is defined.

13.30.84 setNumber()

Sets a parameter to a Number value.

Syntax

```
void setNumber(
   unsigned int paramIndex,
   const Number &val);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2, and so on.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
`val`	The parameter value.

13.30.85 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

```
void setObject(
   unsigned int paramIndex,
   PObject* val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The object containing the input parameter value.
`sqltyp`	The SQL type name of the object to be set.

13.30.86 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

```
void setPrefetchMemorySize(
   unsigned int bytes);
```

Parameter	Description
`bytes`	Number of bytes used for storing data fetched during each server round trip.

13.30.87 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setPrefetchRowCount(
   unsigned int rowCount);
```

Parameter	Description
`rowCount`	Number of rows to fetch for each round trip to the server.

13.30.88 setRef()

Sets the value of a parameter to a reference. A `Ref<T>` instance is implicitly converted to a `RefAny` object during this call.

Syntax	Description

`void setRef(`
`  unsigned int paramIndex,`
`  const RefAny &refAny);`
	Sets the value of a parameter to a reference.

`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`.

`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].

13.30.89 setRowCountsOption()

Set `flag` to `TRUE` to enable return of DML row counts per iteration when invoking getDMLRowCounts().

This option should be set before the statement executes. By default, the DML row counts per iteration are not returned.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setRowCountsOption(
   bool flag);
```

Parameter	Description
`flag`	TRUE to return DML row counts, FALSE otherwise

13.30.90 setRowid()

Sets a `Rowid` bytes array for a bind position.

Syntax

```
void setRowid(
   unsigned int paramIndex,
   const Bytes &val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The parameter value.

13.30.91 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

```
void setSQL(
   const string &sql);
```

Parameter	Description
`sql`	Any SQL statement.

13.30.92 setSQLUString()

Associate an SQL statement with this object. Unicode support: the client `Environment` should be initialized in OCCIUTF16 mode.

Syntax

```
void setSQLUString(
   const UString &sql);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
`sql`	A SQL statement in same character set as the connection source of the statement.

13.30.93 setString()

Sets a parameter for a specified index.

Syntax

```
void setString(
   unsigned int paramIndex,
   const string &val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The parameter value.

13.30.94 setTimestamp()

Sets a parameter to a `Timestamp` value.

Syntax

```
void setTimestamp(
   unsigned int paramIndex,
   const Timestamp &val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The parameter value.

13.30.95 setUInt()

Sets a parameter to a C++ unsigned int value.

Syntax

```
void setUInt(
   unsigned int paramIndex,
   unsigned int val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
`val`	The parameter value.

13.30.96 setUString()

Sets a parameter for a specified index; globalization enabled.

Syntax

```
void setUString(
   unsigned int paramIndex,
   const UString &val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2, and so on.
`val`	The parameter value.

13.30.97 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	Description

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector< T > &vect,`
`  const string &schemaName,`
`  const string &typeName);`
	Intended for use on platforms where partial ordering of function templates is not supported, such as Windows NT and z/OS. Multibyte support.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<T* > &vect,`
`  const string &schemaName,`
`  const string &typeName);`
	Intended for use on platforms where partial ordering of function templates is supported. Multibyte support.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<BDouble> &vect`
`  const string &sqltype);`
	Sets a `BDouble` vector.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Bfile> &vect,`
`  const string &schemaName,`
`  const string &typeName);`
	Sets a `const Bfile` vector; multibyte support.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Bfile> &vect,`
`  const UString &schemaName,`
`  const UString &typeName);`
	Sets a `const BFile` vector; UTF16 support.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<BFloat> &vect`
`  const string &sqltype);`
	Sets a `BFloat` vector.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Blob> &vect,`
`  const string &schemaName,`
`  const string &typeName);`
	Sets a `const Blob` vector; multibyte support.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Blob> &vect,`
`  const UString &schemaName,`
`  const UString &typeName);`
	Sets a `const Blob` vector; UTF16 support.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Bytes> &vect,`
`  const string &sqltype);`
	Sets a `const Bytes` vector.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Bytes> &vect,`
`  const string &schemaName,`
`  const string &typeName);`
	Sets a `const Bytes` vector; multibyte support.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Bytes> &vect,`
`  const Ustring &schemaName,`
`  const Ustring &typeName);`
	Sets a `const Bytes` vector; UTF16 support.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Clob> &vect,`
`  const string &schemaName,`
`  const string &typeName);`
	Sets a `const Clob` vector; multibyte support.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Clob> &vect,`
`  const UString &schemaName,`
`  const UString &typeName);`
	Sets a `const Clob` vector; UTF16 support.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Date> &vect,`
`  const string &schemaName,`
`  const string &typeName);`
	Sets a `const Date` vector; multibyte support.

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Date> &vect,`
`  const UString &schemaName,`
`  const UString &typeName);`
	Sets a `const Date` vector; UTF16 support.

`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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`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; UTF16 support

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void setVector(`
`  Statement *stmt,`
`  unsigned int paramIndex,`
`  const vector<Timestamp> &vect,`
`  const UString &schemaName,`
`  const UString &typeName);`
	Sets a `const Timestamp` vector; UTF16 support.

`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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`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.

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`	The vector to be set.
`sqltype`	Sqltype of the collection in the database. For example, `CREATE TYPE num_coll AS VARRAY OF NUMBER`. And the column/parameter type is `num_coll`. The sqltype would be `num_coll`.
`schemaName`	Name of the schema used
`typeName`	Type

13.30.98 setVectorOfRefs()

Sets a parameter to a vector; is necessary when the type is a collection of `REF`s or nested tables of `REF`s.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`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 &sqltype);`	Used for multibyte support.
`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 not supported, such as Windows NT and z/OS.
`template <class T> void setVector( Statement *stmt, unsigned int paramIndex, const vector<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.
`sqltype`	Sqltype of the parameter or column. Use setVectorOfRefs() instead of specialized function setVector() for `Ref<T>`.
`schemaName`	Name of the schema used
`typeName`	Type

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.30.99 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 getCurrentIteration() 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.

Syntax

```
Status status() const;
```

13.31 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

Attribute	Options
`Status`	• `READY_FOR_READ` indicates that the `Stream` is ready for read operations • `READY_FOR_WRITE` indicates that the `Stream` is ready for write operations • `INACTIVE` indicates that the `Stream` is not available for ready or write operations

Table 13-46 Summary of Stream Methods

Method	Summary
readBuffer()	Reads the stream and returns the amount of data read from the Stream object.
readLastBuffer()	Reads last buffer from Stream.
writeBuffer()	Writes data from buffer to the stream.
writeLastBuffer()	Writes the last data from buffer to the stream.
status()	Returns the current status of the stream.

13.31.1 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
virtual int readBuffer(
   char *buffer,
   unsigned int size) = 0;
```

Parameter	Description
`buffer`	Pointer to data buffer; must be allocated and freed by caller.
`size`	Specifies the number of bytes to be read.

13.31.2 readLastBuffer()

Reads the last buffer from the `Stream`. It can also be called top 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

```
virtual int readLastBuffer(
   char *buffer,
   unsigned int size) = 0;
```

Parameter	Description
`buffer`	Pointer to data buffer; must be allocated and freed by caller.
`size`	Specifies the number of bytes to be read.

13.31.3 writeBuffer()

Writes data from buffer to the stream. The amount of data is determined by `size`.

Syntax

```
virtual void writeBuffer(
   char *buffer,
   unsigned int size) = 0;
```

Parameter	Description
`buffer`	Pointer to data buffer.
`size`	Specifies the number of `chars` to be written.

13.31.4 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
virtual void writeLastBuffer(
   char *buffer,
   unsigned int size) = 0;
```

Parameter	Description
`buffer`	Pointer to data buffer.
`size`	Specifies the number of bytes to be written.

13.31.5 status()

Returns the current `Status`, as defined in Table 13-45 .

Syntax

```
virtual Status status() const;
```

13.32 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

Attribute	Options
`Presentation`	• `PRES_DEFAULT` indicates that the event notification should be in default format. • `PRES_XML` indicates that the event notification should be in XML format.
`Protocol`	• `PROTO_CBK` indicates that the client receives notifications through the default system protocol. • `PROTO_MAIL` indicates that the client receives notifications through e-mail, like `xyz@oracle.com`. The database does not check if the e-mail is valid. • `PROTO_SERVER` indicates that the client receives notifications through an invoked PL/SQL procedure in the database, like `schema.procedure`. The subscriber must have the appropriate permissions on the procedure. • `PROTO_HTTP` indicates that the client receives notifications through an HTTP URL, like `http://www.oracle.com:80`. The database does not check if the URL is valid.
`Namespace`	• `NS_ANONYMOUS` indicates that the registrations are made in an anonymous namespace. • `NS_AQ` indicates that the registrations are made in the Oracle Database Advanced Queuing namespace.

Table 13-48 Summary of Subscription Methods

Method	Summary
Subscription()	`Subscription` class constructor.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-48 (Cont.) Summary of Subscription Methods

Method	Summary
getCallbackContext()	Retrieves the callback context.
getDatabaseServersCount()	Retrieves the number of database servers in which the client is interested for the registration.
getDatabaseServerNames()	Returns the names of all the database servers where the client registered an interest for notification.
getNotifyCallback()	Returns the pointer to the registered callback function.
getPayload()	Retrieves the payload that has been set on the `Subscription` object before posting.
getSubscriptionName()	Retrieves the name of the `Subscription`.
getSubscriptionNamespace()	Retrieves the namespace of the `Subscription`.
getRecipientName()	Retrieves the name of the `Subscription` recipient.
getPresentation()	Retrieves the notification presentation mode.
getProtocol()	Retrieves the notification protocol.
isNull()	Determines if the `Subscription` is `NULL`.
operator=()	Assignment operator for `Subscription`.
setCallbackContext()	Registers a callback function for OCI protocol.
setDatabaseServerNames()	Specifies the database server distinguished names from which the client receives notifications.
setNotifyCallback()	Specifies the context passed to user callbacks
setNull()	Specifies the `Subscription` object to `NULL` and frees the memory associated with the object.
setSubscriptionName()	Specifies the name of the subscription.
setSubscriptionNamespace()	Specifies the namespace in which the subscription is used.
setPayload()	Specifies the buffer content of the notification.
setRecipientName()	Specifies the name of the recipient of the notification.
setPresentation()	Specifies the presentation mode in which the client receives notifications.
setProtocol()	Specifies the protocol in which the client receives notifications.
setSubscriptionName()	Specifies the name of the subscription.
setSubscriptionNamespace()	Specifies the namespace where the subscription is used.
setRecipientName()	Specifies the name of the recipient of the notification.

13.32.1 Subscription()

`Subscription` class constructor.

Syntax	Description
`Subscription (` `  const Environment *env );`	Creates a `Subscription` within a specified `Environment`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`Subscription(` `  const Subscription& sub);`	Copy constructor.

Syntax

```
Subscription(const Subscription& sub);
```

Parameter	Description
`env`	The `Environment`.
`sub`	The original `Subscription`.

13.32.2 getCallbackContext()

Retrieves the callback context.

Syntax

```
void* getCallbackContext() const;
```

13.32.3 getDatabaseServersCount()

Returns the number of database servers in which the client is interested for the registration.

Syntax

```
unsigned int getDatabaseServersCount() const;
```

13.32.4 getDatabaseServerNames()

Returns the names of all the database servers where the client registered an interest for notification.

Syntax

```
vector<string> getDatabaseServerNames() const;
```

13.32.5 getNotifyCallback()

Returns the pointer to the callback function registered for this `Subscription`.

Syntax

```
unsigned int (*getNotifyCallback() const)(
   Subscription& sub,
   NotifyResult *nr);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
sub	The Subscription.
nr	The NotifyResult.

13.32.6 getPayload()

Retrieves the payload that has been set on the Subscription object before posting.

Syntax

```
Bytes getCPayload() const;
```

13.32.7 getSubscriptionName()

Retrieves the name of the subscription.

Syntax

```
string getSubscriptionName() const;
```

13.32.8 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.

Syntax

```
Namespace getSubscriptionNamespace() const;
```

13.32.9 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;
```

13.32.10 getPresentation()

Retrieves the presentation mode in which the client receives notifications. Valid Presentation values are defined in Table 13-47.

Syntax

```
Presentation getPresentation() const;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

13.32.11 getProtocol()

Retrieves the protocol in which the client receives notifications. Valid `Protocol` values are defined in Table 13-47.

Syntax

```
Protocol getProtocol() const;
```

13.32.12 isNull()

Returns `TRUE` if `Subscription` is `NULL` or `FALSE` otherwise.

Syntax

```
bool isNull() const;
```

13.32.13 operator=()

Assignment operator for `Subscription`.

Syntax

```
void operator=(
   const Subscription& sub);
```

Parameter	Description
`sub`	The original `Subscription`.

13.32.14 setCallbackContext()

Registers a notification callback function when the protocol is `PROTO_CBK`, as defined in Table 13-47. Context registration is also included in this call.

Syntax

```
void setCallbackContext(
   void *ctx);
```

Parameter	Description
`ctx`	The context set.

13.32.15 setDatabaseServerNames()

Specifies the list of database server distinguished names from which the client receives notifications.

Syntax

```
void setDatabaseServerNames(
   const vector<string>& dbsrv);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
dbsrv	The list of database distinguished names

13.32.16 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

```
void setNotifyCallback(
   unsigned int (*callback)(
                   Subscription& sub,
                   NotifyResult *nr));
```

Parameter	Description
callback	The user callback function.
sub	The Subscription object.
nr	The NotifyResult object.

13.32.17 setNull()

Sets the Subscription object to NULL and frees the memory associated with the object.

Syntax

```
void setNull();
```

13.32.18 setPayload()

Sets the buffer content that corresponds to the payload to be posted to the Subscription.

Syntax

```
void setPayload(
   const Bytes& payload);
```

Parameter	Description
payload	Content of the notification.

13.32.19 setPresentation()

Sets the presentation mode in which the client receives notifications.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setPresentation(
   Presentation pres);
```

Parameter	Description
`pres`	Presentation mode, as defined in Table 13-47.

13.32.20 setProtocol()

Sets the `Protocol` in which the client receives event notifications, as defined in Table 13-47.

Syntax

```
void setProtocol(
   Protocol prot);
```

Parameter	Description
`prot`	Protocol mode

13.32.21 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

```
void setSubscriptionName(
   const string& name);
```

Parameter	Description
`name`	Subscription name.

13.32.22 setSubscriptionNamespace()

Sets the namespace where the subscription is used. The subscription name must be consistent with its namespace. Default value is `NS_AQ`.

Syntax

```
void setSubscriptionNamespace(
   Namespace nameSpace);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
nameSpace	Namespace in which the subscription is used, as defined in Table 13-47.

13.32.23 setRecipientName()

Sets the name of the recipient of the notification.

Syntax

```
void setRecipientName(
   const string& name);
```

Parameter	Description
name	Name of the notification recipient.

13.33 Timestamp Class

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 10*g* 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

Field	Type	Minimum Value	Maximum value
year	int	-4713	9999
month	unsigned int	1	12
day	unsigned int	1	31
hour	unsigned int	0	23
min	unsigned int	0	59

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-49 (Cont.) Fields of Timestamp and Their Legal Ranges

Field	Type	Minimum Value	Maximum value
`sec`	unsigned int	`0`	`61`
`tzhour`	int	`-12`	`14`
`tzmin`	int	`-59`	`59`

Table 13-50 Summary of Timestamp Methods

Method	Summary
Timestamp()	`Timestamp` class constructor.
fromText()	Sets the time stamp from the values provided by the string.
getDate()	Gets the date from the `Timestamp` object.
getTime()	Gets the time from the `TimeStamp` object.
getTimeZoneOffset()	Returns the time zone hour and minute offset value.
intervalAdd()	Returns a `Timestamp` object with value (this + interval).
intervalSub()	Returns a `Timestamp` object with value (this - interval).
isNull()	Checks if `Timestamp` is `NULL`.
operator=()	Simple assignment.
operator==()	Checks if a and b are equal.
operator!=()	Checks if a and b are not equal.
operator>()	Checks if a is greater than b.
operator>=()	Checks if a is greater than or equal to b.
operator<()	Checks if a is less than b.
operator<=()	Checks if a is less than or equal to b.
setDate()	Sets the year, month, day components contained for this timestamp.
setNull()	Sets the value of Timestamp to `NULL`
setTime()	Sets the day, hour, minute, second and fractional second components for this timestamp.
setTimeZoneOffset()	Sets the hour and minute offset for time zone.
subDS()	Returns a `IntervalDS` representing this - `val`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Table 13-50 (Cont.) Summary of Timestamp Methods

Method	Summary
subYM()	Returns a `IntervalYM` representing this - `val`.
toText()	Returns a `string` representation for the timestamp in the format specified.

13.33.1 Timestamp()

`Timestamp` class constructor.

Syntax	Description

`Timestamp(`
`  const Environment *env,`
`  int year=1,`
`  unsigned int month=1,`
`  unsigned int day=1,`
`  unsigned int hour=0,`
`  unsigned int min=0,`
`  unsigned int sec=0,`
`  unsigned int fs=0,`
`  int tzhour=0,`
`  int tzmin=0);`
	Returns a default `Timestamp` object. Time components are understood to be in the specified time zone.

`Timestamp();`	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=() .

`Timestamp(`
`  const Environment *env,`
`  int year,`
`  unsigned int month,`
`  unsigned int day,`
`  unsigned int hour,`
`  unsigned int min,`
`  unsigned int sec,`
`  unsigned int fs,`
`  const string &timezone);`
	Multibyte support. The timezone can be passed as region, "US/Eastern", or as an offset from GMT, "+05:30". If an empty string is passed, then the time is considered to be in the current session's time zone. Used for constructing values for `TIMESTAMP WITH LOCAL TIME ZONE` types.

`Timestamp(`
`  const Environment *env,`
`  int year,`
`  unsigned int month,`
`  unsigned int day,`
`  unsigned int hour,`
`  unsigned int min,`
`  unsigned int sec,`
`  unsigned int fs,`
`  const UString &timezone);`
	UTF16 (`UString`) support. The timezone can be passed as region, "US/Eastern", or as an offset from GMT, "+05:30". If an empty string is passed, then the time is considered to be in the current session's time zone. Used for constructing values for `TIMESTAMP WITH LOCAL TIME ZONE` types.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description
`Timestamp(` `  const Timestamp &src);`	Copy constructor.

Parameter	Description
`year`	Year component.
`month`	Month component.
`day`	Day component.
`hour`	Hour component.
`minute`	Minute component.
`second`	Second component.
`fs`	Fractional second component.
`tzhour`	Time zone difference hour component.
`tzmin`	Timezone difference minute component.
`src`	The original `Timezone`.

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:

```
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.getDate(yr, mth, day);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

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 ts1;
ts1.fromText("01:16:17.12 04/03/1825", "hh:mi:ssxff 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:mi:ss [tzh:tzm]",0);
   if(reft<ts                        //compare timestamps
      IntervalDS ds=reft.subDS(ts);  //get difference between timestamps
}
```

13.33.2 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`

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax	Description

`void fromText(`
`  const string &timestmpStr,`
`  const string &fmt,`
`  const string &nlsParam = "",`
`  const Environment *env = NULL);`
	Sets `Timestamp` object to value represented by a `string`.

`void fromText(`
`  const UString &timestmpStr,`
`  const UString &fmt,`
`  const UString &nlsParam,`
`  const Environment *env = NULL);`
	Sets `Timestamp` object to value represented by a `UString`; globalization enabled.

Parameter	Description
`timestmpStr`	The timestamp `string` or `UString` to be converted to a `Timestamp` object.
`fmt`	The format string.
`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`.
`env`	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.

13.33.3 getDate()

Returns the year, month and day values of the `Timestamp`.

Syntax

```
void getDate(
   int &year,
   unsigned int &month,
   unsigned int &day) const;
```

Parameter	Description
`year`	Year component.
`month`	Month component.
`day`	Day component.

13.33.4 getTime()

Returns the hour, minute, second, and fractional second components

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void getTime(
   unsigned int &hour,
   unsigned int &minute,
   unsigned int &second,
   unsigned int &fs) const;
```

Parameter	Description
`hour`	Hour component.
`minute`	Minute component.
`second`	Second component.
`fs`	Fractional second component.

13.33.5 getTimeZoneOffset()

Returns the time zone offset in hours and minutes.

Syntax

```
void getTimeZoneOffset(
   int &hour,
   int &minute) const;
```

Parameter	Description
`hour`	Time zone hour.
`minute`	Time zone minute.

13.33.6 intervalAdd()

Adds an interval to timestamp.

Syntax	Description
`const Timestamp intervalAdd(` `  const IntervalDS& val) const;`	Adds an IntervalDS interval to the timestamp.
`const Timestamp intervalAdd(` `  const IntervalYM& val) const;`	Adds an IntervalYM interval to the timestamp.

Parameter	Description
`val`	Interval to be added.

Copyright © 1999, 2025, Oracle and/or its affiliates.

13.33.7 intervalSub()

Subtracts an interval from a timestamp and returns the result as a timestamp. Returns a `Timestamp` with the value of this - `val`.

Syntax	Description
`const Timestamp intervalSub( const IntervalDS& val) const;`	Subtracts an `IntervalDS` interval to the timestamp.
`const Timestamp intervalsUB( const IntervalYM& val) const;`	Subtracts an `IntervalYM` interval to the timestamp.

Parameter	Description
`val`	Interval to be subtracted.

13.33.8 isNull()

Returns `TRUE` if `Timestamp` is `NULL` or `FALSE` otherwise.

Syntax

```
bool isNull() const;
```

13.33.9 operator=()

Assigns a given timestamp object to this object.

Syntax

```
Timestamp & operator=(
   const Timestamp &src);
```

Parameter	Description
`src`	Value to be assigned.

13.33.10 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

```
bool operator==(
   const Timestamp &first,
   const Timestamp &second);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
first	First timestamp to be compared.
second	Second timestamp to be compared.

13.33.11 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

```
bool operator!=(
   const Timestamp &first,
   const Timestamp &second);
```

Parameter	Description
first	First timestamp to be compared.
second	Second timestamp to be compared.

13.33.12 operator>()

Returns TRUE if first is greater than second, FALSE otherwise. If either is NULL then FALSE is returned.

Syntax

```
bool operator>(
   const Timestamp &first,
   const Timestamp &second);
```

Parameter	Description
first	First timestamp to be compared.
second	Second timestamp to be compared.

13.33.13 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.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
bool operator>=(
   const Timestamp &first,
   const Timestamp &second);
```

Parameter	Description
first	First timestamp to be compared.
second	Second timestamp to be compared.

13.33.14 operator<()

Returns TRUE if first is less than second, FALSE otherwise. If either a or b is NULL then FALSE is returned.

Syntax

```
bool operator<(
   const Timestamp &first,
   const Timestamp &second);
```

Parameter	Description
first	First timestamp to be compared.
second	Second timestamp to be compared.

13.33.15 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

```
bool operator<=(
   const Timestamp &first,
   const Timestamp &second);
```

Parameter	Description
first	First timestamp to be compared.
second	Second timestamp to be compared.

13.33.16 setDate()

Sets the year, month, day components contained for this timestamp

Copyright © 1999, 2025, Oracle and/or its affiliates.

Syntax

```
void setDate(
   int year,
   unsigned int month,
   unsigned int day);
```

Parameter	Description
`year`	Year component. Valid values are -4713 through 9999.
`month`	Month component. Valid values are 1 through 12.
`day`	Day component. Valid values are 1 through 31.

13.33.17 setNull()

Sets the timestamp to `NULL`.

Syntax

```
void setNull();
```

13.33.18 setTime()

Sets the day, hour, minute, second and fractional second components for this timestamp.

Syntax

```
void setTime(
   unsigned int hour,
   unsigned int minute,
   unsigned int second,
   unsigned int fs);
```

Parameter	Description
hour	Hour component. Valid values are 0 through 23.
minute	Minute component. Valid values are 0 through 59.
second	Second component. Valid values are 0 through 59.
fs	Fractional second component.

13.33.19 setTimeZoneOffset()

Sets the hour and minute offset for time zone.

Syntax

```
void setTimeZoneOffset(
   int hour,
   int minute);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
hour	Time zone hour. Valid values are -12 through 12.
minute	Time zone minute. Valid values are -59 through 59.

13.33.20 subDS()

Computes the difference between this timestamp and the specified timestamp and return the difference as an `IntervalDS`.

Syntax

```
const IntervalDS subDS(
   const Timestamp& val) const;
```

Parameter	Description
val	Timestamp to be subtracted.

13.33.21 subYM()

Computes the difference between timestamp values and return the difference as an `IntervalYM`.

Syntax

```
const IntervalYM subYM(
   const Timestamp& val) const;
```

Parameter	Description
`val`	`Timestamp` to be subtracted.

13.33.22 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`

Copyright © 1999, 2025, Oracle and/or its affiliates.

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
`fmt`	The format string.
`fsprec`	The precision for the fractional second component of `Timestamp`.
`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`.

Copyright © 1999, 2025, Oracle and/or its affiliates.

14

Vector Support in Oracle AI Database

This chapter describes the Oracle AI Database vector support in OCCI.

Starting with Oracle AI Database 26ai vectors as a datatype is introduced to represent semi-structured and unstructured data for AI and machine learning semantics. Vectors are represented as a series of numbers with a specified length and number type.

The vector elements can be represented by the following data types:

- INT8_t : This corresponds to an int8 (8-bit signed integer) data type in C++
- FLOAT32 :This corresponds to a float in C++
- FLOAT64 :This corresponds to a double type in C++
- BINARY :Although C++ does not support `binary` data type, `uin8_t` is used to support binary data. Each `uint8_t` value has 8 bits (binary values). BINARY vectors must have a dimension that is a multiple of 8.

> **See Also**
>
> Overview of Oracle AI Vector Search

Topics:

- Oracle Database Vector Support in OCCI
- Sparse Vectors
- Utility Function for SparseVector

14.1 Oracle Database Vector Support in OCCI

This section describes the Oracle Database vector support functions in OCCI.

Topics:

- Binding Vectors in a Statement Handle
- PL/SQL Out-Binds for Vectors
- Defining Vectors from a ResultSet of a Query
- MetaData of Vectors

14.1.1 Binding Vectors in a Statement Handle

This section describes the functions introduced for binding the vector data to a statement handle.

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.1.1.1 setInt8Vector()

Sets the value of an Oracle Database Vector data type in `int8(ub1)` format from a C++ vector value of type `int8_t`.

Syntax

```
void setInt8Vector(unsigned int paramIndex,vector<int8_t> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	Specifies the vector to be set.

14.1.1.2 setFloatVector()

Sets the value of an Oracle Database Vector data type in `IEEE FLOAT32` format from a C++ vector value of type `float`.

Syntax

```
void setFloatVector(unsigned int paramIndex,vector<float> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	Specifies the vector to be set.

14.1.1.3 setDoubleVector()

Sets the value of an Oracle Database Vector data type in format `IEEE FLOAT64` from a C++ vector of type double.

Syntax

```
void setDoubleVector(unsigned int paramIndex,vector<double> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	Specifies the vector to be set.

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.1.1.4 setBinaryVector()

Sets the value of an Oracle Database Vector data type in binary format from a C++ vector of type `uint8_t`.

> **Note**
>
> You can only insert binary vectors with dimensions that are a multiple of 8.

Syntax

```
void setBinaryVector(unsigned int paramIndex,vector<uint8_t> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	Specifies the vector to be set.

The following sample code snippet, binds the vector `myData` as a `FLOAT32` to position 1 in `stmt`. On executing the statement further, it is inserted in the database table.

```
Statement *stmt;
stmt = conn->createStatement ("INSERT INTO test_vecflt values(:1,:2)");
vector<float> myData({1.4,1.2,3.5});
stmt->setInt(1,10);
stmt->setFloatVector(2,myData);
stmt->executeUpdate();
```

14.1.2 PL/SQL Out-Binds for Vectors

This section describes the functions used to get the output from PL/SQL procedures or functions through out-binds in a statement handle.

14.1.2.1 getInt8Vector()

Gets the value of an Oracle Database Vector data type in `int8(ub1)` format from a C++ vector value of type `int8_t` format.

Syntax

```
void getInt8Vector(unsigned int paramIndex,vector<int8_t> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	References to the vector (`OUT` parameter) into which the values must be inserted.

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.1.2.2 getFloatVector()

Gets the value of an Oracle Database Vector in `IEEE FLOAT32` format from a C++ vector value of type `float`.

Syntax

```
void getFloatVector(unsigned int paramIndex,vector<float> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	References to the vector (`OUT` parameter) into which the values must be inserted.

14.1.2.3 getDoubleVector()

Gets the value of an Oracle Database Vector data type in `IEEE FLOAT64` format from a C++ vector value of type double.

Syntax

```
void getDoubleVector(unsigned int paramIndex,vector<double> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	References to the vector (`OUT` parameter) into which the values must be inserted.

14.1.2.4 getBinaryVector()

Gets the value of an Oracle Database Vector data type in binary format inside a C++ vector of type `uint8_t`.

Syntax

```
void getBinaryVector(unsigned int paramIndex,vector<uint8_t>&vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	References to the vector (`OUT` parameter) into which the values must be inserted.

Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> Currently, the dimension of binary vectors is a multiple of 8. That is, `8*vec.size()`where, `vec` is a `stl vector<uint8_t>`.

The following code snippet can be used for out binds to obtain a vector:

```
Statement *stmt;
stmt = conn->createStatement ("BEGIN occivec_demo(:1) END;");
//Disclaimer: Binary literals introduced in C++14 and above, used here for
descriptive purposes.
vector<uint8_t> myInBind({0b00100011,0b01000011,0b10101011,0b01001001});
stmt->setBinaryVector(1,myInOutBind);
stmt->stmtExecute();
stmt->getBinaryVector(1,myInOutBind);
```

This obtains the data fetched from `INOUT` bind at position 1 in `myInOutBind`.

In case of flexible dimensions, there may be a case where the size of the vector is not the same as the dimension of the Database vector. In such a case, `stmt->getBinaryVector()` will always have a return value indicating the dimension of the `OUT` vector.

```
int dimension = stmt->getBinaryVector(1,myInOutBind);
//printing the bits
for(int i=0;i<dimension;i++)
    cout<<((v[i/8]>>(8-(i%8)))&1;
```

The following code snippet can be used for out binds to obtain a vector:

```
Statement *stmt;
stmt = conn->createStatement ("BEGIN occivec_demo(:1,:2,:3) END;");
vector<double> myInBind({1.4,1.2,3.5});
vector<double> myInOutBind;
vector<double> myOutBind;
stmt->setInt(1,myInBind);
stmt->setDoubleVector(2,myInOutBind);
stmt->registerOutParam(3,OCCI_SQLT_VEC);
stmt->stmtExecute();
stmt->getDoubleVector(2,myInOutBind);
stmt->getDoubleVector(3,myOutBind);
```

14.1.3 Defining Vectors from a ResultSet of a Query

This section describes the functions introduced in the ResultSet class to obtain the column values of type vector into C++ stl vector containers.

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.1.3.1 getInt8Vector()

Gets the column value of an Oracle Database Vector data type in `int8(ub1)` format as a C++ vector value of type `int8_t`.

Syntax

```
void getInt8Vector(unsigned int paramIndex,vector<int8_t> &vec);
```

Parameter	Description
`colIndex`	Specifies the column index, first column is 1, second is 2, and so on.
`vec`	References to the vector (OUT parameter) into which the values must be inserted.

14.1.3.2 getFloatVector()

Gets the column value of an Oracle Database Vector data type in `IEEE FLOAT32` format from a C++ vector value of type `float`.

Syntax

```
void getFloatVector(unsigned int paramIndex,vector<float> &vec);
```

Parameter	Description
`colIndex`	Specifies the column index, first column is 1, second is 2, and so on.
`vec`	References to the vector (OUT parameter) into which the values must be inserted.

14.1.3.3 getDoubleVector()

Gets the column value of an Oracle Database Vector data type in `IEEE FLOAT64` format from a C++ vector value of type `double`.

Syntax

```
void getDoubleVector(unsigned int paramIndex,vector<double> &vec);
```

Parameter	Description
`colIndex`	Specifies the column index, first column is 1, second is 2, and so on.
`vec`	References to the vector (OUT parameter) into which the values must be inserted.

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.1.3.4 getBinaryVector()

Gets the column value of an Oracle Database Vector data type in binary format from a C++ vector value of type `uint8_t`.

Syntax

```
void getBinaryVector(unsigned int paramIndex,vector<uint8_t> &vec);
```

Parameter	Description
`colIndex`	Specifies the column index, first column is 1, second is 2, and so on.
`vec`	References to the vector (OUT parameter) into which the values must be inserted.

The following code snippet can be used in ResultSet to obtain a vector:

```
Statement stmt;
stmt = conn->createStatement ("SELECT * FROM TEST_VECFLT");
ResultSet *rs = stmt->getResultSet();
vector<float> myResult;
stmt->stmtExecute();
rs->getFloatVector(2,myResult);
```

The type of vector data in the result depends on the function called. For example, `getFloatVector` corresponds to `FLOAT32` format.

14.1.3.5 getString()

Gets the column value of an Oracle Database vector of any format as a C++ string.

Syntax

```
string getString(unsigned int paramIndex);
```

Parameter	Description
`colIndex`	Specifies the column index, first column is 1, second is 2, and so on.

The following sample code snippet can be used in ResultSet to obtain a vector:

```
Statement stmt;
stmt = conn->createStatement ("SELECT id,embedding FROM TEST_SPARSEVECFLT");
ResultSet *rs = stmt->getResultSet();
stmt->stmtExecute();
int id = rs->getInt(1);
string vec = rs->getString(2);
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

14.1.4 MetaData of Vectors

A new data type `OCCI_SQLT_VEC` is introduced for supporting MetaData in describe functionality.

The following column attributes specific to vector column have been added to MetaData:

- **ATTR_VECTOR_DIMENSION** : Information of vector dimensions
- **ATTR_VECTOR_DATA_FORMAT**: Type of data format stored in a vector column
- **ATTR_VECTOR_PROPERTY**: Information of the data flag

The following code snippet is used to get the column metadata for a vector:

```
vector<MetaData> v1;
MetaData metaData = conn->getMetaData("TEST_VECDBL");
v1 = metaData.getVector(MetaData::ATTR_LIST_COLUMNS);
for(int i=0; i < v1.size(); i++)
{
    MetaData md = v1[i];
    cout << " Column Name :" << md.getString(MetaData::ATTR_NAME)
<< endl;
    cout << " Data Type :"   << md.getInt(MetaData::ATTR_DATA_TYPE)
<< endl;
    cout << " Dimension :"   << md.getUInt(MetaData::ATTR_VECTOR_DIMENSION)
<< endl;
}
```

14.1.4.1 Example

```
#include <sys/types.h>
#include <fstream>
#include <iostream>
#include <occi.h>
#include <cstdlib>
#include <string.h>

using namespace oracle::occi;
using namespace std;

static  char *username=strdup("scott");
static  char *password=strdup("tiger");
static  char *dbname=strdup("inst1");

class occivec
{
  private:

  Environment *env;
  Connection *conn;
  Statement *stmt;
  ResultSet *rs;
  public:

  occivec ()
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  {
    env = Environment::createEnvironment (Environment::DEFAULT);
    conn = env->createConnection (username, password, dbname);
  }

  ~occivec ()
  {
    env->terminateConnection (conn);
    Environment::terminateEnvironment (env);
  }

  void displayAllRows (string tableName)
  {
    cout<<"--- Displaying rows of "<<tableName<<"---"<<endl;
    string sqlStmt = "select * from "+tableName;
    stmt = conn->createStatement (sqlStmt);
    ResultSet *rset = stmt->executeQuery ();
    try{
    while (rset->next ())
    {
      cout << "ID: " << rset->getInt (1) << endl;
      vector<double> ans;
      rset->getDoubleVector(2,ans);
      cout << "Embedding: ";
      for(auto i:ans)cout<<i<<" ";cout<<endl;
    }
    }catch(SQLException ex)
    {
     cout<<"Exception thrown for displayAllRows"<<endl;
     cout<<"Error number: "<<  ex.getErrorCode() << endl;
     cout<<ex.getMessage() << endl;
    }

    stmt->closeResultSet (rset);
    conn->terminateStatement (stmt);
    cout<<endl;
  }

  void testproc ()
  {
    cout << "--- Testing Procedure Binds ---" << endl;
    vector<double> myInBind = {10.1,20.2,30.3,40.4};
    vector<double> myInOutBind = {50.5,60.6,70.7,80.8};
    vector<double> myOutBind;
    try{
      cout << "callproc - invoking a PL/SQL procedure having IN, OUT and
IN/OUT ";
      stmt = conn->createStatement("BEGIN occivec_demo(:1,:2,:3); END;");
      stmt->setDoubleVector(1,myInBind);
      stmt->setDoubleVector (2,myInOutBind);
      stmt->registerOutParam (3, OCCI_SQLT_VEC);
      stmt->executeUpdate ();
      stmt->getDoubleVector(2,myInOutBind);
      stmt->getDoubleVector(3,myOutBind);
      cout << "Printing the INOUT & OUT parameters:" << endl;
      cout<< "IN/OUT vector:"<<endl;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
      for(auto i:myInOutBind)cout<<i<<" ";cout<<endl;
      cout<< "OUT vector:"<<endl;
      for(auto i:myOutBind)cout<<i<<" ";cout<<endl;
      conn->terminateStatement (stmt);
      cout << endl;
    }catch(SQLException ex)
    {
      cout<<"Exception thrown for testproc"<<endl;
      cout<<"Error number: "<<  ex.getErrorCode() << endl;
      cout<<ex.getMessage() << endl;
    }
  }

  void insertElement ()
  {
    cout << "--- Inserting in a vector table ---" << endl;
    int num = 1;
    vector<double> myData = {10,20.30,30,40.50};
    vector<double> myResult;
    try{
        stmt = conn->createStatement ("INSERT INTO test_vecint
values(:1,:2)");
        stmt->setInt(1,num);
        stmt->setDoubleVector(2,myData);
        stmt->executeUpdate();
        conn->terminateStatement (stmt);
        cout << endl;
    }catch(SQLException ex)
    {
        cout<<"Exception thrown for insertElement"<<endl;
        cout<<"Error number: "<<  ex.getErrorCode() << endl;
        cout<<ex.getMessage() << endl;
    }
  }

  void describe_table ()
  {
    cout << "--- Describing the table - TEST_VECDBL ---" << endl;
    vector<MetaData> v1;
    MetaData metaData = conn->getMetaData("TEST_VECDBL");
    v1 =  metaData.getVector(MetaData::ATTR_LIST_COLUMNS);
    for(int i=0;i<v1.size();i++)
    {
        MetaData md = v1[i];
        cout << " Column Name :" << (md.getString(MetaData::ATTR_NAME)) <<
endl;
        cout << " Data Type :" << (printType
(md.getInt(MetaData::ATTR_DATA_TYPE))) << endl;
        cout << " Dimension :"<< md.getUInt(MetaData::ATTR_VECTOR_DIMENSION)
<< endl;
        cout << endl;
    }
    cout << endl;
  }

  string printType (int type)
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
  {
    switch (type)
    {
      case OCCI_SQLT_NUM : return "NUMBER";
                           break;
      case OCCI_SQLT_VEC:  return "VECTOR";
                           break;
      default: return to_string(type);
    }
  } // End of printType (int)

};

int main (void)
{
  try{
    cout << "Demo Program for OCCI Vector" << endl;
    occivec *demo = new occivec ();
    demo->insertElement();
    demo->describe_table();
    demo->testproc();
    demo->displayAllRows("TEST_VECDBL");
    delete (demo);
  }
  catch (SQLException ex){
    cout << ex.getMessage() << endl;
  }
  cout << "done" << endl;
}
```

14.2 Sparse Vectors

This section describes the functions to work with the SparseVectors. Sparse vectors are preferred as they reduce the cost of memory.

Currently C++ has no implicit sparse-vector support, so SparseVector class is introduced as a data structure.

> **Note**
>
> Sparse vectors follow a 0-based indexing conforming with the current vector implementations.

Topics:

- SparseVector Constructors
- Binding Sparse Vectors in a Statement Handle
- PL/SQL out-binds for Sparse Vectors
- Defining Sparse Vectors from a ResultSet of a Query

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.2.1 SparseVector Constructors

This section describes the SparseVector constructors.

14.2.1.1 SparseVector()

SparseVector function initializes a SparseVector of required type that fixes the type of values it uses.

This is the default constructor:

```
SparseVector<T> mySparseVector();
```

This function can be used in your library to construct a sparse vector as follows:

```
SparseVector<float> mySparseVector;
```

14.2.1.2 SparseVector(uint32_t size,vector<uint32_t> indices,vector<T> values)

This function initializes a SparseVector of required size and non_zeroes values with respective indices.

Copying these vectors to the vector members of our SparseVector class.

> **Note**
>
> The size must strictly be greater than or equal to the length of the indices and values vector. Also, ensure that the length of indices and the values are same.

Syntax

```
SparseVector<T> mySparseVector(uint32_t size, vector<uint32_t> indices,
vector<A>
    values);
```

Parameter	Description
`size`	Specifies the total size of the sparse vector.
`indices`	Specifies the indices of non-zero values of the sparse vector.
`values`	Specifies the values of the non-zero indices in the sparse vector.

This function can be used in your library to construct a sparse vector as follows:

```
SparseVector<float> mySparseVector(45,{1,5,40},{1.3,2.5,4554}};
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

14.2.1.3 SparseVec(uint32_t size,pair<vector<uint32_t>,vector<T>> &vecp) Ref constructor

This function is more efficient or a faster constructor.

You do not need to copy the vectors in your own class vector members. Instead, in this constructor, we can utilize references to the vectors inside the pair given as your argument. This means that all the changes made to the user vectors are reflected in your SparseVector and vice-versa.

Syntax

```
SparseVector<T> myCopySparseVector(uint32_t size,
pair<vector<uint32_t>,vector<T>>
    &vecp);
```

Parameter	Description
size	Specifies the total size of the sparse vector.
vecp	Specifies the pair of vectors containing the index and the value of the vectors respectively.

The following code snippet shows how this function is used to construct a sparse vector from an existing sparse vector:

```
pair<vector<uint32_t>,vector<float>> vecp({1,3,5},{2,4,6});
SparseVector<float> mySparseVector(100,vecp); // now we initialize this does
no copies of the vectors inside our pair

string sqlStmt = "SELECT * FROM DEMO_TAB"; //Table has one row with two
columns (id:2,SparseVector:10,{0,2,4},{3,5,7})
stmt = conn->createStatement (sqlStmt);
rset = stmt->executeQuery ();
while(rset->next())
{
    cout << "ID: " << rset->getInt (1) << endl;
    rset->getSparseVector(2, mySparseVector);
    //each time we do getSparseVector our vecp directly gets modified to the
sparse-vector we recieved.
    //We can check this by printing each vectors of vecp.
    for(auto i:vecp.first)
        cout<<i<<" ";
    cout<<endl;
    // this will print 0,2,4 as our indices in SparseVector of our table
    for(auto i:vecp.second)
        cout<<i<<" ";
    cout<<endl;
    // this will print 3,5,7 as our indices in SparseVector of our table
}
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

> **Note**
>
> The pair we used for initialization that is, vecp should not go out of scope when you are using your mySparseVector for the database operations. Otherwise, your SparseVector is also invalidated.

14.2.2 Binding Sparse Vectors in a Statement Handle

The functions described in this section are introduced for binding SparseVector data to a statement handle.

> **Note**
>
> Functionality of sparse vector and dense vector bind/define functions:
> We can insert using `SparseVector/std::vector` regardless of the table having dense/sparse vector columns. However, we can only fetch SparseVector from sparse vector columns and dense vectors `(std::vector)` from dense vector columns.

14.2.2.1 setInt8Vector()

Binds the value to a Sparse Vector in `INT8` format from an OCCI SparseVector of type `int8_t`.

Syntax

```
void setInt8Vector(unsigned int paramIndex, SparseVector<int8_t> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	Specifies the SparseVector to be set.

14.2.2.2 setFloatVector()

Binds the value to a Sparse Vector `IEEE FLOAT32` format from an OCCI SparseVector of type `float`.

Syntax

```
void setFloatVector(unsigned int paramIndex, SparseVector<float> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	Specifies the SparseVector to be set.

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.2.2.3 setDoubleVector()

Binds the value to a Sparse Vector in `IEEE FLOAT64` format from an OCCI SparseVector of type `double`.

Syntax

```
void setDoubleVector(unsigned int paramIndex, SparseVector<double> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	Specifies the SparseVector to be set.

14.2.2.4 setBinaryVector()

Sets the value of an Oracle Database Vector data type in binary format from a C++ vector of type `uint8_t`

> **Note**
>
> You can only insert vectors with dimension that are a mutliple of 8.

Syntax

```
void setBinaryVector(unsigned int paramIndex, SparseVector<uint8_t> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	Specifies the SparseVector to be set.

The following code snippet binds the SparseVector `mySparseVector` of `FLOAT32` format to position 1 in `stmt`. Further executing the statement, inserts it in the database table:

```
Statement *stmt;
stmt = conn->createStatement ("INSERT INTO test_sparsevecflt values(:1,:2)");
vector<uint32_t> inds = {0, 5, 9};
vector<float> vals = {1.5, 2.3, 4.6};
SparseVector<float> mySparseVector(10, inds, vals);
stmt->setInt(1,10);
stmt->setFloatVector(2, mySparseVector);
stmt->executeUpdate();
```

14.2.3 PL/SQL out-binds for Sparse Vectors

The functions described in this section, can be used to get output from PLSQL procedures or functions through out-binds in the statement handle.

Copyright © 1999, 2025, Oracle and/or its affiliates.

14.2.3.1 getInt8Vector()

Gets the value of an Oracle Database sparse vector data type in `int8(ub1)` format as an OCCI SparseVector of type `int8_t`.

Syntax

```
void getInt8Vector(unsigned int paramIndex, SparseVector<int8_t> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	References to the SparseVector (`OUT` parameter) into which the values must be inserted.

14.2.3.2 getFloatVector()

Gets the value of an Oracle Database sparse vector data type in `IEEE FLOAT32` format as an OCCI SparseVector of type `float`.

Syntax

```
void getFloatVector(unsigned int paramIndex, SparseVector<float> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	References to the SparseVector (`OUT` parameter) into which the values must be inserted.

14.2.3.3 getDoubleVector()

Gets the value of an Oracle Database sparse vector data type in `IEEE FLOAT64` format as an OCCI SparseVector of type doubles.

Syntax

```
void getDoubleVector(unsigned int paramIndex, SparseVector<double> &vec);
```

Parameter	Description
`paramIndex`	Specifies the parameter index; first parameter is 1, second is 2, and so on.
`vec`	References the SparseVector (`OUT` parameter) into which the values must be inserted.

The following sample code snippet uses this function for out binds to obtain a vector:

```
Statement *stmt;
stmt = conn->createStatement ("BEGIN occivec_demo(:1,:2,:3) END;");
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
pair<vector<uint32_t>, vector<double>> myInPair = {{3, 7, 8}, {2.6, 7.6,
1001.1}};
pair<vector<uint32_t>, vector<double>> myInOutPair = {{0, 4, 9}, {3, 5.7,
2001}};
pair<vector<uint32_t>, vector<double>> myOutPair;
SparseVector<uint32_t> myInBind(9, myInPair);
SparseVector<uint32_t> myInOutBind(10, myInOutPair);
SparseVector<uint32_t> myOutBind(0, myOutPair);
stmt->setDoubleVector(1, myInBind);
stmt->setDoubleVector(2, myInOutBind);
stmt->registerOutParam(3, OCCI_SQLT_VEC);
stmt->stmtExecute();
stmt->getDoubleVector(2, myInOutBind);
stmt->getDoubleVector(3, myOutBind);
```

The preceding code snippet gets the data obtained from INOUT bind at position 2 into `myInOutBind` SparseVector as well as the `OUT` bind at position 3 in `myOutBind`.

14.2.4 Defining Sparse Vectors from a ResultSet of a Query

This section describes the functions introduced in ResultSet class to obtain the column values of type SparseVector into C++ `stl` vector containers.

14.2.4.1 getInt8Vector()

Gets the column value of an Oracle Database sparse vector datatype in `int8(ub1)` as an OCCI SparseVector of type `int8_t`.

Syntax

```
void getInt8Vector(unsigned int paramIndex, SparseVector<int8_t> &vec);
```

Parameter	Description
`colIndex`	Specifies the column index, first column is 1, second is 2, and so on.
`vec`	References to the SparseVector (`OUT` parameter) into which the values must be inserted.

14.2.4.2 getFloatVector()

Gets the column value of an Oracle Database sparse vector data type in `IEEE FLOAT32` format as an OCCI SparseVector of type float.

Syntax

```
void getFloatVector(unsigned int paramIndex, SparseVector<float> &vec);
```

Parameter	Description
`colIndex`	Specifies the column index, first column is 1, second is 2, and so on.

Copyright © 1999, 2025, Oracle and/or its affiliates.

Parameter	Description
vec	References to the SparseVector (OUT parameter) into which the values must be inserted.

14.2.4.3 getDoubleVector()

Gets the column value of an Oracle Database vector data type in IEEE FLOAT64 format as an OCCI SparseVector of type double.

Syntax

```
void getDoubleVector(unsigned int paramIndex, SparseVector<double> &vec);
```

Parameter	Description
colIndex	Specifies the column index, first column is 1, second is 2, and so on.
vec	References to the SparseVector (OUT parameter) into which the values must be inserted.

The following code snippet shows how this function can be used in the ResultSet to obtain a Sparse Vector:

```
Statement stmt;
stmt = conn->createStatement ("SELECT * FROM TEST_SPARSEVECFLT");
ResultSet *rs = stmt->getResultSet();
pair<vector<uint32_t>,vector<float>> myResult;
SparseVector mySparseVector(0, myResult);
stmt->stmtExecute();
rs->getFloatVector(2, myResult);
```

The type of SparseVector data in result is decided depending on the vector type provided. That is, SparseVector<float> corresponds to FLOAT32 format.

14.2.4.4 getString()

Gets the column value of an Oracle Database sparse vector data type of any format as a C++ string.

Syntax

```
string getString(unsigned int paramIndex);
```

Parameter	Description
colIndex	Specifies the column index, first column is 1, second is 2, and so on.

The following code snippet shows how to use this function in ResultSet to obtain a vector:

```
Statement stmt;
stmt = conn->createStatement ("SELECT id,embedding FROM TEST_SPARSEVECFLT");
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

```
ResultSet *rs = stmt->getResultSet();
stmt->stmtExecute();
int id = rs->getInt(1);
string vec = rs->getString(2);
```

14.3 Utility Function for SparseVector

This section describes a member function of the SparseVector class and returns a dense form of the vector stored by your SparseVector.

14.3.1 toDenseVec()

Returns the dense vector form of an OCCI SparseVector of type <T>.

Syntax

```
vector<T> toDense();
```

> **Note**
>
> Here T is the desired type of vector value.

The following code snippet shows how you can use this function in SparseVector to obtain a dense vector:

```
SparseVector<float> mySparseVector(10,{0,5,9},{0.5,5.5,9.5});
vector<float> myDenseVector = mySparseVector.toDense();
for(auto i:myDenseVector)
    cout<<i<<" ";
cout<<endl;
```


Copyright © 1999, 2025, Oracle and/or its affiliates.

Index

A

ADR, *15*
  ADRC utility, *17*
  base location, *15*
ADR Command Interpreter, *15*
ADRCI, *15*, *17*
Agent class, *5*
  methods, *5*
AnyData class, *8*
  methods, *8*
  supported data type, *8*
  usage examples, *8*
application managed data buffering, *8*
application-provided serialization, *7*
associative access
  overview, *9*
atomic null, *18*
attributes, *8*
automatic diagnostic repository (ADR), *15*
automatic serialization, *7*

B

BatchSQLException
  methods, *17*
BatchSQLException class, *17*
Bfile class, *18*
  methods, *18*
BFILEs
  external data type, *9*
bind operations
  in bind operations, *8*
  out bind operations, *7*
Blob class, *24*
  methods, *24*
BLOBs
  external data type, *10*
Bytes class, *33*
  methods, *33*

C

callable statements, *15*
  with arrays as parameters, *15*
CASE OTT parameter, *6*
CHAR
  external data type, *10*
classes
  Agent class, *5*
  AnyData class, *8*
  BatchSQLException class, *17*
  Bfile class, *18*
  Blob class, *24*
  Bytes class, *33*
  Clob class, *35*
  Connection class, *47*
  ConnectionPool class, *60*
  Consumer class, *65*
  Date class, *73*
  Environment class, *83*
  IntervalDS class, *96*
  IntervalYM class, *107*
  Listener class, *116*
  Map class, *118*
  Message class, *119*
  Metadata class, *127*
  NotifyResult class, *139*
  Number class, *140*
  PObject class, *159*
  Producer class, *165*
  Ref class, *170*
  RefAny class, *175*
  ResultSet class, *20*, *178*
  SQLException class, *198*
  StatelessConnectionPool class, *201*
  Statement class, *211*
  Stream class, *260*
  Subscription class, *262*
  Timestamp class, *269*
Client Result Cache, *19*
  hint, *19*
Clob class, *35*
  methods, *35*
CLOBs
  external data type, *10*
CODE OTT parameter, *7*
collections
  working with, *16*
committing a transaction, *27*
complex object retrieval
  complex object, *14*

Copyright © 1999, 2025, Oracle and/or its affiliates.

complex object retrieval *(continued)*
depth level, *14*
implementing, *15*
overview, *14*
prefetch limit, *14*
root object, *14*
complex objects, *14*
prefetching, *16*
retrieving, *15*
CONFIG OTT parameter, *7*
configuration files
and the OTT utility, *2*
connecting to a database, *1*
Connection class, *47*
methods, *47*
connection pool
createConnectionPool method, *4*
creating, *3*, *4*
connection pooling, *3*
transparent application failover, *3*
ConnectionPool class, *60*
methods, *60*
Consumer class, *65*
methods, *65*
control statements, *5*

D

data buffering, *8*
data conversion
Date, *22*
DATE data type, *22*
internal data type, *19*
Interval, *22*
INTERVAL data type, *22*
LOB data type, *21*
LOBs, *21*
Timestamp, *22*
TIMESTAMP data type, *22*
data type
AnyData, *8*
external data type, *1*, *4*
internal data type, *2*
internal data types, *1*
OTT mappings, *19*
overview, *1*
data types, *1*
database
connecting to, *1*
database resident connection pooling, *9*
administration, *10*
using, *11*
DATE
external data type, *10*
data conversion, *22*
Date class, *73*
methods, *73*
usage examples, *73*
DDL statements
executing, *12*
depth level, *14*
DML statements
executing, *12*

E

elements, *4*
embedded objects, *2*
creating, *2*
fetching, *17*
prefetching, *17*
Environment class, *83*
methods, *83*
ERRTYPE OTT parameter, *7*
examples
Date class, *73*
IntervalDS class, *96*
IntervalYM class, *107*
Number class, *140*
exception handling, *29*
executing SQL queries, *20*
executing statements dynamically, *22*
external data type, *8*
BFILE, *9*
BLOB, *10*
CHAR, *10*
CHARZ, *10*
CLOB, *10*
DATE, *10*
FLOAT, *11*
INTEGER, *12*
INTERVAL DAY TO SECOND, *12*
INTERVAL YEAR TO MONTH, *12*
LONG, *13*
LONG RAW, *13*
LONG VARCHAR, *13*
LONG VARRAW, *13*
NCLOB, *13*
NUMBER, *13*
OCCI BFILE, *14*
OCCI BLOB, *14*
OCCI BYTES, *14*
OCCI CLOB, *15*
OCCI DATE, *15*
OCCI INTERVALDS, *15*
OCCI INTERVALYM, *15*
OCCI NUMBER, *15*
OCCI POBJECT, *15*
OCCI REF, *16*
OCCI REFANY, *16*
OCCI STRING, *16*

Copyright © 1999, 2025, Oracle and/or its affiliates.

external data type *(continued)*
OCCI TIMESTAMP, *16*
OCCI VECTOR, *16*
RAW, *16*
REF, *17*
ROWID, *17*
STRING, *17*
TIMESTAMP, *17*
TIMESTAMP WITH LOCAL TIME ZONE, *17*
TIMESTAMP WITH TIME ZONE, *18*
UNSIGNED INT, *18*
VARCHAR, *18*
VARCHAR2, *18*
VARNUM, *18*
VARRAW, *19*

F

fault diagnosability, *15*
Fault Diagnosability
disabling, *19*
fields
IntervalDS class, *96*
IntervalYM class, *107*
Timestamp fields, *269*
FLOAT
external data type, *11*

H

HFILE OTT parameter, *7*

I

identity column metadata, *2*
index-organized table, *4*
INTEGER
external data type, *12*
internal data type, *2*
INTERVAL DAY TO SECOND
external data type, *12*
INTERVAL YEAR TO MONTH
external data type, *12*
IntervalDS class, *96*
fields, *96*
methods, *96*
usage examples, *96*
IntervalYM class, *107*
fields, *107*
methods, *107*
usage examples, *107*
INTYPE file
structure of, *15*
INTYPE OTT parameter, *8*

L

Listener class, *116*
methods, *116*
LOBs
external data type
data conversion, *21*
LONG
external data type, *13*
LONG RAW
external data type, *13*
LONG VARCHAR
external data type, *13*

M

manipulating object attributes, *12*
Map class, *118*
methods, *118*
Message class, *119*
methods, *119*
metadata
argument and result attributes, *16*
attribute groupings, *3*
argument and result attributes, *3*
collection attributes, *3*
column attributes, *3*
database attributes, *3*
list attributes, *3*
package attributes, *3*
parameter attributes, *3*
procedure, function, and subprogram attributes, *3*
schema attributes, *3*
sequence attributes, *3*
synonym attributes, *3*
table and view attributes, *3*
type attribute attributes, *3*
type attributes, *3*
type method attributes, *3*
attributes, *8*
code example, *4*
collection attributes, *14*
column attributes, *15*
database attributes, *18*
describing database objects, *3*
list attributes, *18*
overview, *1*
package attributes, *10*
parameter attributes, *8*
procedure, function, and subprogram attributes, *10*
schema attributes, *18*
sequence attributes, *15*
synonym attributes, *15*
table and view attributes, *9*

Copyright © 1999, 2025, Oracle and/or its affiliates.

metadata *(continued)*
type attribute attributes, *12*
type attributes, *11*
type methods attributes, *13*
MetaData class, *127*
methods, *127*
methods, *8*
Agent methods, *5*
AnyData methods, *8*
BatchSQLException methods, *17*
Bfile methods, *18*
Blob methods, *24*
Bytes methods, *33*
Clob methods, *35*
Connection methods, *47*
ConnectionPool methods, *60*
Consumer methods, *65*
createConnection method, *3*
createConnectionPool method, *4*
createEnvironment method, *2*
createProxyConnection method, *5*
createStatement method, *12*
Date methods, *73*
Environment class, *83*
execute method, *12*
executeArrayUpdate method, *12*, *10*
executeQuery method, *12*
IntervalDS methods, *96*
IntervalYM class, *107*
Listener methods, *116*
Map methods, *118*
Message methods, *119*
MetaData class, *127*
NotifyResult methods, *139*
Number class, *140*
PObject methods, *159*
Producer methods, *165*
Ref methods, *170*
RefAny methods, *175*
ResultSet methods, *178*
setDataBuffer method, *9*
SQLException methods, *198*
StatelessConnectionPool, *201*
Statement, *211*
Stream methods, *260*
Subscription methods, *262*
terminateConnection method, *3*
terminateEnvironment method, *3*
terminateStatement method, *14*
Timestamp methods, *269*
modifying rows iteratively, *11*

N

navigational access
overview, *10*
NCLOBs
external data type, *13*
NEEDS_STREAM_DATA status, *23*, *24*
nonprocedural elements, *4*
nonreferenceable objects, *2*
NotifyResult class, *139*
methods, *139*
nullness, *18*
NUMBER
external data type, *13*
Number class, *140*
methods, *140*
usage examples, *140*

O

object cache, *8*
flushing, *8*
object mode, *7*
object programming
overview, *1*
using OCCI, *1*
object references
see also REF, *11*
using, *18*
Object Type Translator utility
see OTT utility, *8*
object types, *8*
objects
access using SQL, *10*
attributes, *8*
client-side, *8*
dirty, *12*
environment, *9*
flushing, *12*
freeing, *18*
in OCCI, *2*
inserting, *10*
interfaces, *9*
associative, *9*
navigational, *9*
manipulating attributes, *12*
marking, *12*
Metadata Class, *10*
methods, *8*
modifying, *10*
object cache, *8*
object types, *8*
pinned, *11*
pinning, *8*, *11*
recording database changes, *13*
run-time environment, *9*
transparent application failover, *3*
OCCI
benefits, *2*
building applications, *3*

Copyright © 1999, 2025, Oracle and/or its affiliates.

OCCI *(continued)*
functionality, *4*
object mode, *7*
overview, *2*
special SQL terms, *7*
OCCI classes
Agent class, *5*
AnyData class, *8*
BatchSQLException class, *17*
Bfile class, *18*
Blob class, *24*
Bytes class, *33*
Clob class, *35*
Connection class, *47*
ConnectionPool class, *60*
Consumer class, *65*
Data class, *73*
Environment class, *83*
IntervalDS class, *96*
IntervalYM class, *107*
Listener class, *116*
Map class, *118*
Message class, *119*
MetaData class, *127*
NotifyResult class, *139*
Number class, *140*
PObject class, *159*
Producer class, *165*
Ref class, *170*
RefAny class, *175*
ResultSet class, *178*
SQLException class, *198*
StatelessConnectionPool class, *201*
Statement class, *211*
Stream class, *260*
Subscription class, *262*
Timestamp class, *269*
OCCI environment
connection pool, *3*
creating, *1*
opening a connection, *2*, *3*
scope, *1*, *2*
terminating, *1*
OCCI program
example of, *22*
OCCI program development, *5*
operational flow, *7*
program structure, *6*
OCCI types
data conversion, *1*
optimizing performance, *22*
setting prefetch count, *22*
Oracle Connection Manager in Traffic Director Mode, *12*
OTT parameter TRANSITIVE, *10*
OTT parameters
CASE, *6*
CODE, *7*
CONFIG, *7*
ERRTYPE, *7*
HFILE, *7*
INTYPE, *8*
OUTTYPE, *8*
SCHEMA_NAMES, *9*
USERID, *12*
where they appear, *12*
OTT utility
benefits, *10*
creating types in the database, *2*
default name mapping, *23*
description, *10*
parameters, *5*
using, *2*
out bind variables, *6*
OUTTYPE OTT parameter, *8*

P

parameterized statements, *14*
performance
optimizing
executeArrayUpdate method, *10*
setDataBuffer method, *9*
performance tuning, *1*
application managed data buffering, *8*
array fetch using next() method, *11*
connection sharing, *5*
data buffering, *8*
reading and writing multiple LOBs, *7*
shared server environments, *6*
thread safety, *6*
thread safety, *6*
transparent application failover, *1*
persistent objects, *2*
creating, *4*
standalone objects, *2*
types
embedded objects, *2*
nonreferenceable objects, *2*
referenceable objects, *2*
standalone objects, *2*
pinning objects, *8*, *11*
PL/SQL
out bind variables, *6*
overview, *6*
pluggable databases
OCCI support for, *3*
PObject class, *159*
methods, *159*
prefetch count
set, *22*

Copyright © 1999, 2025, Oracle and/or its affiliates.

prefetch limit, *14*
PREPARED status, *23*
procedural elements, *4*
Producer class, *165*
  methods, *165*
proxy connections, *5*
  using createProxyConnection method, *5*

Q

queries, *6*
  how to specify, *21*

R

RAW
  external data type, *16*
REF
  external data type, *17*
  retrieving a reference to an object, *11*
Ref class, *170*
  methods, *170*
RefAny class, *175*
  methods, *175*
referenceable objects, *2*
relational programming
  using OCCI, *1*
RESULT_SET_AVAILABLE status, *23*
ResultSet class, *20*, *178*
  methods, *178*
root object, *14*
ROWID
  external data type, *17*
rows
  iterative modification, *11*
  modify, *11*

S

SCHEMA_NAMES OTT parameter, *9*
shared connections
  using, *5*
shared server environments
  application-provided serialization, *7*
  automatic serialization, *7*
  concurrency, *7*
  thread safety, *6*
    implementing, *6*
SQL statements
  control statements, *5*
  DML statements, *6*
  processing of, *5*
  queries, *6*
  types
    callable statements, *14*, *15*
    parameterized statements, *14*
SQL statements *(continued)*
    types *(continued)*
    standard statements, *14*
SQLException class, *198*
  methods, *198*
sqlnet.ora, *19*
standalone objects, *2*
  creating, *2*
standard statements, *14*
StatelessConnectionPool class, *201*
  methods, *201*
statement caching, *27*
Statement class, *211*
  methods, *211*
statement handles
  creating, *12*
  reusing, *13*
  terminating, *14*
status
  NEEDS_STREAM_DATA, *23*, *24*
  PREPARED, *23*
  RESULT_SET_AVAILABLE, *23*
  STREAM_DATA_AVAILABLE, *23*, *24*
  UNPREPARED, *23*
  UPDATE_COUNT_AVAILABLE, *23*, *24*
Stream class, *260*
  methods, *260*
STREAM_DATA_AVAILABLE status, *23*, *24*
streamed reads, *16*
streamed writes, *16*
STRING
  external data type, *17*
Subscription class, *262*
  methods, *262*
substitutability, *20*

T

table
  index-organized, *4*
thread safety, *6*
  implementing, *6*
TIMESTAMP
  external data type, *17*
Timestamp class
  methods, *269*
TIMESTAMP WITH LOCAL TIME ZONE
  external data type, *17*
TIMESTAMP WITH TIME ZONE
  external data type, *18*
transient objects, *2*, *3*
  creating, *3*, *4*
TRANSITIVE OTT parameter, *10*
transparent application failover, *1*
  connection pooling, *3*
  objects, *3*

Copyright © 1999, 2025, Oracle and/or its affiliates.

transparent application failover *(continued)*
using, *2*
type inheritance, *19*, *21*

U

UNPREPARED status, *23*
UNSIGNED INT
external data type, *18*
UPDATE_COUNT_AVAILABLE status, *23*, *24*
USERID OTT parameter, *12*

V

values
in context of this document, *4*
in object applications, *4*
VARCHAR
external data type, *18*
VARCHAR2
external data type, *18*
VARNUM
external data type, *18*
VARRAW
external data type, *13*, *19*

X

XA library, *1*

Copyright © 1999, 2025, Oracle and/or its affiliates.