Oracle® C++ Call Interface

Programmer's Guide

10*g* Release 1 (10.1)

Part No. B10778-01

December 2003

Oracle C++ Call Interface Programmer's Guide, 10*g* Release 1 (10.1)

Part No. B10778-01

Primary Author: Roza Leyderman

Contributors: Sandeepan Banerjee, Subhranshu Banergee, Kalyanji Chintakayala, Krishna Itikarlapalli, Shankar Iyer, Maura Joglekar, Ravi Kasamsetty, Srinath Krishnaswamy, Shoaib Lari, Geoff Lee, Chetan Maiya, Rekha Vallam

Contents

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle C++ Call Interface Programmer's Guide, 10*g* Release 1 (10.1)

Part No. B10778-01

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

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

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

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

  Oracle Corporation

  Server Technologies Documentation

  500 Oracle Parkway, Mailstop 4op11

  Redwood Shores, CA 94065 USA

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

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

Preface

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

This preface contains these topics:

Audience

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

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

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

Organization

This document contains:

Chapter 1, "Introduction to OCCI"

This chapter introduces you to OCCI and describes special terms and typographical conventions that are used in describing OCCI.

Chapter 2, "Relational Programming"

This chapter gives you the basic concepts needed to develop an OCCI program. It discusses the essential steps each OCCI program must include, and how to retrieve and understand error messages.

Chapter 3, "Object Programming"

This chapter provides an introduction to the concepts involved when using OCCI to access objects in an Oracle database server. The chapter includes a discussion of basic object concepts and object navigational access, and the basic structure of object-relational applications.

Chapter 4, "Datatypes"

This chapter discusses Oracle internal and external data types, and necessary data conversions.

Chapter 5, "Metadata"

This chapter discusses how to use the `MetaData()` method to obtain information about schema objects and their associated elements.

Chapter 6, "Object Type Translator Utility"

This chapter discusses the use of the Object Type Translator (OTT) to convert database object definitions to C++ representations for use in OCCI applications.

Chapter 7, "Globalization and Unicode Support"

This chapter discusses the Unicode and globalization support for OCCI applications.

Chapter 8, "Oracle Streams Advanced Queuing"

This chapter describes the Oracle Streams support for asynchronous messages in OCCI applications, otherwise known as Advanced Queuing.

Chapter 9, "Oracle XA Library"

This chapter describes the Oracle XA Library and contains a brief example of application development with XA support.

Chapter 10, "OCCI Application Programming Interface"

This chapter describes the OCCI classes and methods for C++.

Related Documentation

For more information, see these Oracle resources:

- OCCI product information page for OCCI white papers, additional examples, and so on, at http://otn.oracle.com/tech/oci/occi/index.html
- Discussion forum for all OCCI related information is at http://forums.oracle.com/forums/forum.jsp?forum=168
- Demos at `$ORACLE_HOME/rdbms/demo`
- *Oracle Database Concepts*
- *Oracle Database SQL Reference*
- *Oracle Database Application Developer's Guide - Object-Relational Features*
- *Oracle Database New Features*
- *Oracle Call Interface Programmer's Guide*
- *Oracle Database Administrator's Guide*
- *Oracle Streams Advanced Queuing User's Guide and Reference*

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.

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

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

```
http://otn.oracle.com/membership/
```

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

```
http://otn.oracle.com/documentation/
```

Conventions

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

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

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

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width font)`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

Convention	Meaning	Example
`lowercase monospace (fixed-width font)`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase monospace (fixed-width font) italic`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

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

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

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

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`

|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}`
		
		`[COMPRESS | NOCOMPRESS]`

Convention	Meaning	Example

`. . .`	Horizontal ellipsis points indicate either:
	▪ That we have omitted parts of the code that are not directly related to the example
	▪ That you can repeat a portion of the code
		`CREATE TABLE ... AS` *`subquery`*`;`
		
		`SELECT` *`col1`*`,` *`col2`*`, ... ,` *`coln`* `FROM employees;`

.
.
.
	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	`//process information in buffer`
		`.`
		`.`
		`blob.close();`

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);`
		`acct    CONSTANT NUMBER(4) := 3;`

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

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

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

Documentation Accessibility

Accessibility of Code Examples in Documentation

Accessibility of Links to External Web Sites in Documentation

What's New in Oracle C++ Call Interface?

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

New Features for 10*g* Release 1 (10.1)

The following features are new to this release:

- OCCI Support for Windows NT on page 10-5 for accessing collections of `Refs` in ResultSet Class and Statement Class, in Chapter 10, "OCCI Application Programming Interface"
- This release provides OCCI libraries for Microsoft CRT debugging and for developing applications with Microsoft Visual C++ 7.0 (.NET). Please see the Windows platform Readme for details on supported compiler versions.
- NATIVE DOUBLE Datatype on page 4-20 in Chapter 4, "Datatypes" supports IEEE754Double
- NATIVE FLOAT Datatype on page 4-20 in Chapter 4, "Datatypes" supports IEEE754Float
- Instant Client Feature on page 1-5 in Chapter 1, "Introduction to OCCI"
- Enhancements in the base PObject Class on page 10-194 in Chapter 6, "Object Type Translator Utility"; OTT C++ classes must be re-generated after migrating to this release
- Stateless Connection Pooling on page 2-6 and StatelessConnectionPool Class on page 10-249
- Globalization and Unicode support in the new Chapter 7, "Globalization and Unicode Support"

- Oracle Streams Advanced Queuing in the new Chapter 8, "Oracle Streams Advanced Queuing"
- XA Compliance support in the new Chapter 9, "Oracle XA Library"
- "Statement Caching" on page 2-26 in Chapter 2, "Relational Programming"
- Array Pinning for Objects: Section "Persistent Objects" on page 3-2 in Chapter 3, "Object Programming".
- Section "Migrating C++ Applications Using OCCI" on page 3-12 in Chapter 3, "Object Programming"
- Timestamp Class in Chapter 10, "OCCI Application Programming Interface"behavior is enhanced:
  - Users no longer need to convert to GMT when using Timestamp() constructor on page 10-329, or in methods setDate() on page 10-338 and setTime() on page 10-339
  - New constructors that support timezone information as string or `UString`(Unicode) enable users to pass a region name, such as "US/Eastern", as a timezone. These provide daylight savings(DST) support. Using an empty string, "", constructs a timestamp in the local timezone.
  - New support for all three `TIMESTAMP` types in the database, for both relational and objects access: `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE`.

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
- Instant Client Feature
- Processing of SQL Statements
- Overview of PL/SQL
- Special OCCI/SQL Terms
- Object Support

Overview of OCCI

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

OCCI provides for:

- High performance applications through the efficient use of system memory and network connectivity
- Scalable applications that can service an increasing number of users and requests

- Comprehensive support for application development by using Oracle database objects, including client-side access to Oracle database objects
- Simplified user authentication and password management
- n-tiered authentication
- Consistent interfaces for dynamic connection management and transaction management in two-tier client/server environments or multitiered environments
- Encapsulated and opaque interfaces

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

Benefits of OCCI

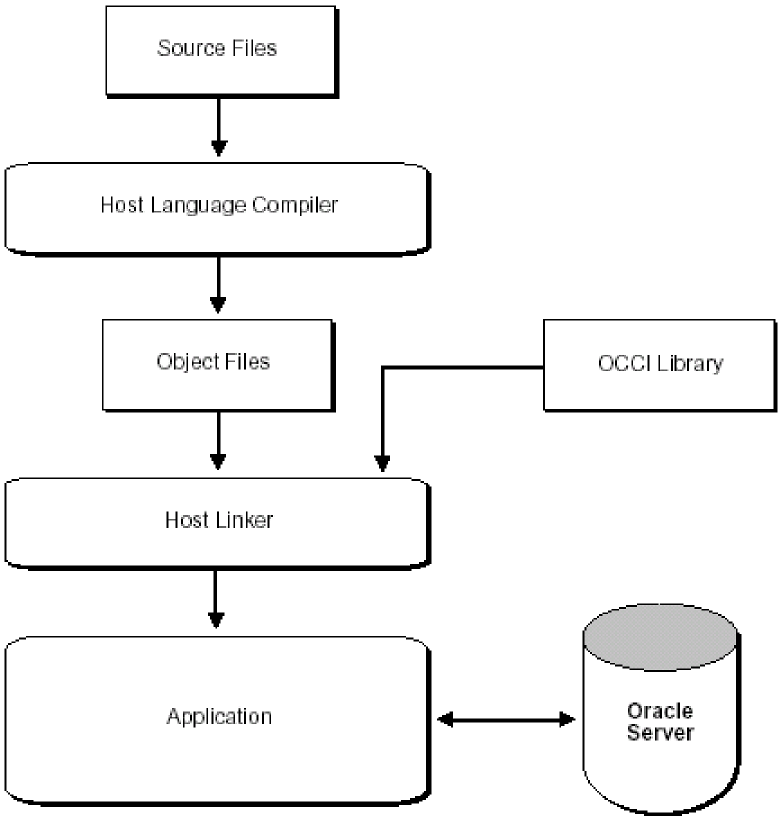
OCCI provides these significant advantages over other methods of accessing an Oracle database:

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

Building an OCCI Application

As Figure 1-1 shows, you compile and link an OCCI program in the same way that you compile and link a nondatabase application.

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.

Functionality of OCCI

OCCI provides the following functionality:

- APIs to design a scalable, multithreaded applications that can support large numbers of users securely
- SQL access functions, for managing database access, processing SQL statements, and manipulating objects retrieved from an Oracle database server

- Datatype 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

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 the procedural capabilities of C++.

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

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

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

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

```
SELECT name FROM employees WHERE empno = :empnumber
```

In this SQL statement, *empnumber* is a placeholder for a value that will be supplied by the application.

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

Instant Client Feature

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

Benefits of Instant Client

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

The OCCI Instant Client capability simplifies OCCI installation. Even though OCCI is independent of `ORACLE_HOME` setting in the Instant Client mode, applications that rely on `ORACLE_HOME` settings can continue operation by setting it to the appropriate value. The activation of the Instant Client mode is only dependent on the ability to load the Instant Client data shared library. In particular, this feature allows interoperability with Oracle applications that use `ORACLE_HOME` for their data, but use a newer release of OCCI. Other components such as shared libraries for network protocols, or security options, must be installed separately.

Installing Instant Client

OCCI requires only four shared libraries (or dynamic link libraries, as they are called on some operating systems) to be loaded by the dynamic loader of the operating system:

- OCI Shared Library (`libociei.so` on Solaris and `oraociei10.dll` on Windows); correct installation of this file determines if you are operating in Instant Client mode

- Client Code Library (`libclnstsh.so.10.1` on Solaris and `oci.dll` on Windows)
- Security Library (`libnnz10.so` on Solaris and `orannzsbb10.dll` on Windows)
- OCCI Library (`libocci.so.10.1` on Solaris and `oraocci10.dll` on Windows)

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

- On Solaris, the `libociei.so` library can be copied from the `$ORACLE_HOME/instantclient` directory. All the other Solaris libraries can be copied from the `$ORACLE_HOME/lib` directory in a full Oracle installation.
- On Windows, the `oraociei10.dll` library can be copied from the `ORACLE_HOME\instantclient` directory. All other Windows libraries can be copied from the `ORACLE_HOME\bin` directory.

If you did not install the database, you can retrieve these libraries by choosing the Instant Client option from the Oracle Universal Installer.

The Instant Client libraries are also available on the Oracle Technology Network (OTN) website at http://otn.oracle.com/tech/oci/occi/index.html.

If these four libraries are accessible through the directory on the OS Library Path variable (`LD_LIBRARY_PATH` on Solaris and `PATH` on Windows), then OCCI operates in the Instant Client mode. In this mode, there is no dependency on `ORACLE_HOME` and none of the other code and data files provided in `ORACLE_HOME` are needed by OCCI (except for the `tnsnames.ora` file as described later).

Note:

1. All libraries must be copied from the same release of `ORACLE_HOME` and should be placed in the same directory
2. On Windows, if `ORACLE_HOME\bin` is also on the `PATH` variable, then in order to operate in the Instant Client mode, the directory containing oraociei10.dll must appear before the `ORACLE_HOME\bin` directory.
3. OCCI Library (`libocci.so.10.1` on Solaris and `oraocci10.dll` on Windows) must be installed in a directory on the OS Library Path variable.

Using Instant Client

The Instant Client feature is designed for running production applications. For development, a full installation is necessary to access OCCI header files, Makefiles, demonstration programs, and so on. In general, all OCCI functionality is available to an application being run in the Instant Client mode, except for server-side external procedures.

Patching Instant Client Shared Libraries on Unix

Because Instant Client is a deployment feature, one of its design objectives is to reducing the number and size of necessary files. Therefore, Instant Client deployment does not include all files for patching shared libraries. You should use the `OPATCH` utility on an `ORACLE_HOME` based full client to patch the Instant Client shared libraries.

After successfully patching Instant Client shared libraries, we recommend that you generate the patch inventory information in `ORACLE_HOME`:

```
opatch lsinventory > opatchinv.out
```

This `opatchinv.out` file contains the record of all patches made, and should be copied to the deployment directory, together with the patched Instant Client libraries.

Regenerating the Data Shared Library

This feature is not available on Windows platforms.

The Instant Client Data Shared Library, `libociei.so`, can be regenerated in an Administrator Install of `ORACLE_HOME`. Executing the following two lines will create a new `libociei.so` file based on current file in `ORACLE_HOME` and place it in the `ORACLE_HOME/instantclient` directory:

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ilibociei
```

Database Connection Names for Instant Client

All Oracle net naming methods that do not require use of `ORACLE_HOME` or `TNS_ADMIN` (to locate configuration files such as tnsnames.ora or sqlnet.ora) work in the Instant Client mode. In particular, the connect string in the `OCIServerAttach()` call can be specified in the following formats:

- A SQL Connect URL string of the form:

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

such as:

```
//myserver111:5521/bjava21
```

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

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

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

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

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

The bequeath adapter or the empty connect strings are not supported. However, an alternate way to use the empty connect string is to set the `TWO_TASK` environment variable on Solaris, or the `LOCAL` variable on Windows, to either a tnsnames.ora entry or an Oracle Net keyword-value pair. If `TWO_TASK` or `LOCAL` is set to a `tnsnames.ora` entry, then the `tnsnames.ora` file must be able to be loaded by `TNS_ADMIN` or `ORACLE_HOME` setting.

Environment Variables for OCCI Instant Client

The `ORACLE_HOME` environment variable no longer determines the location of Globalization Support, CORE, and error message files. An OCCI-only application should not require `ORACLE_HOME` to be set. However, if it is set, it does not have an impact on OCCI's operation. OCCI will always obtain its data from the Data Shared Library. If the Data Shared Library is not available, only then is `ORACLE_HOME` used and a full client installation is assumed. When set, `ORACLE_HOME` should be a valid operating system path name that identifies a directory.

If Dynamic User callback libraries are to be loaded, then as this guide specifies, the callback package has to reside in `ORACLE_HOME/lib` on Solaris or `ORACLE_HOME\bin` on Windows. Therefore, `ORACLE_HOME` should be set in this case.

Environment variables `ORA_NLS33`, `ORA_NLS32`, and `ORA_NLS` are ignored in the Instant Client mode.

In the Instant Client mode, if the ORA_TZFILE variable is not set, then the smaller, default, timezone.dat file from the Data Shared Library is used. If the larger timezlrg.dat file is to be used from the Data Shared Library, then set the ORA_TZFILE environment variable to the name of the file without any absolute or relative path names. That is, on Solaris:

```
setenv ORA_TZFILE timezlrg.dat
```

On Windows:

```
set ORA_TZFILE timezlrg.dat
```

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

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

Processing of SQL Statements

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

- Data definition language (DDL) statements
- Control statements
    - Transaction control statements
    - Connection control statements
    - System control statements
- Data manipulation language (DML) statements
- Queries

DDL Statements

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.

The following is an example of creating and specifying access to a table:

```
CREATE TABLE employees (
   name       VARCHAR2(20),
   ssn        VARCHAR2(12),
   empno      NUMBER(6),
   mgr        NUMBER(6),
   salary     NUMBER(6))

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

DDL statements also allow you to work with objects in the Oracle database, as in the following series of statements which create an object table:

```
CREATE TYPE person_t AS OBJECT (
   name     VARCHAR2(30),
   ssn      VARCHAR2(12),
   address  VARCHAR2(50))

CREATE TABLE person_tab OF person_t
```

Control Statements

OCCI applications treat transaction control, connection control, and system control statements like DML statements.

DML SQL Statements

DML statements can change data in database tables. For example, DML statements are used to perform the following actions:

- Insert new rows into a table
- Update column values in existing rows
- Delete rows from a table
- Lock a table in the database
- Explain the execution plan for a SQL statement

DML statements can require an application to supply data to the database by using input (bind) variables. Consider the following statement:

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

Either this statement can be executed several times with different bind values, or an array insert can be performed to insert several rows in one round-trip to the server.

DML statements also enable you to work with objects in the Oracle database, as in the following example, which inserts an instance of type `person_t` into the object table `person_tab`:

```
INSERT INTO person_tab
   VALUES (person_t('Steve May','123-45-6789','146 Winfield Street'))
```

Queries

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

```
SELECT dname FROM dept
   WHERE deptno = 42
```

Queries can require the program to supply data to the database server by using input (bind) variables, as in the following example:

```
SELECT name
   FROM employees
   WHERE empno = :empnumber
```

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

Overview of PL/SQL

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

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

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

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

A PL/SQL procedure or function can also return an output variable. This is called an **out bind variable**. For example:

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

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.

The following PL/SQL example issues a SQL statement to retrieve values from a table of employees, given a particular employee number. This example also demonstrates the use of placeholders in PL/SQL statements.

```
SELECT ename, sal, comm INTO :emp_name, :salary, :commission
   FROM emp
   WHERE ename = :emp_number;
```

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 need to be specified in your program.

Special OCCI/SQL Terms

This guide uses special terms to refer to the different parts of a SQL statement. Consider the following example of a SQL statement:

```
SELECT customer, address
   FROM customers
   WHERE bus_type = 'SOFTWARE'
   AND sales_volume = :sales;
```

This example contains these parts:

- A SQL **command**: `SELECT`
- Two **select-list items**: `customer` and `address`
- A **table name** in the `FROM` clause: `customers`
- Two **column names** in the `WHERE` clause: `bus_type` and `sales_volume`
- A **literal input value** in the `WHERE` clause: '`SOFTWARE`'
- A **placeholder** for an input (bind) variable in the `WHERE` clause: `:sales`

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

Object Support

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

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

OCCI provides a comprehensive API for programmers seeking to use the Oracle database server's object capabilities. These features can be divided into several major categories:

- Client-side object cache
- Runtime environment for objects
- Associative and navigational interfaces to access and manipulate objects
- Metadata class to describe object type metadata

- Object Type Translator (OTT) utility, which maps internal Oracle schema information to client-side language bind variables

See Also:

- *Oracle Database Concepts* and
- *Oracle Database Application Developer's Guide - Object-Relational Features* for a more detailed explanation of object types and objects

Client-Side Object Cache

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

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

Runtime Environment for Objects

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

- Connecting to an Oracle database server in order 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 system types

Associative and Navigational Interfaces

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

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

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

- Binding object type instances and references as input and output variables of SQL statements and PL/SQL stored procedures
- Executing SQL statements that contain object type instances and references
- Fetching object type instances and references
- Retrieving column values from a result set as objects
- Describing a select-list item of an Oracle object type

OCCI provides a seamless interface for navigating objects, enabling you to manipulate database objects in the same way that you would operate on transient C++ objects. You can dereference the overloaded arrow (`->`) operator on an object reference to transparently materialize the object from the database into the application space.

Metadata Class

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

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

Object Type Translator Utility

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

- **Improving application developer productivity** OTT eliminates the need for application developers to write by hand the host language variables that correspond to schema objects.
- **Maintaining SQL as the data definition language of choice** By providing the ability to automatically map Oracle database schema objects that are created by using SQL to host language variables, OTT facilitates the use of SQL as the data definition language of choice. This in turn allows Oracle to support a consistent, enterprise-wide model of the user's data.
- **Facilitating schema evolution of object types** OTT provides the ability to regenerate included header files when the schema is changed, allowing Oracle applications to support schema evolution.

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

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

- Execution of 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

2 Relational Programming

This chapter describes the basics of developing C++ applications using Oracle C++ Call Interface (OCCI) to work with data stored in relational databases.

This chapter contains these topics:

Connecting to a Database

You have a number of different options with regard to how your application connects to the database. These options are discussed in the following sections:

- Creating and Terminating an Environment
- Opening and Closing a Connection

Creating and Terminating an Environment

All OCCI processing takes place in the context of the `Environment` class. An OCCI environment provides application modes and user-specified memory management functions. The following code example shows how you can create an OCCI environment:

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

All OCCI objects created with the `create`*`xxx`* methods (connections, connection pools, statements) must be explicitly terminated and so, when appropriate, you must also explicitly terminate the environment. The following code example shows how you terminate an OCCI environment.

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

In addition, an OCCI environment should have a scope that is larger than the scope of any objects created in the context of that environment, such as `Bytes`, `BFile`, `Blob`, `Clob`, `IntervalDS`, `IntervalYM`, and `Timestamp`. This concept is demonstrated in the following code example:

```
const string userName = "SCOTT";
const string password = "TIGER";
const string connectString = "";

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

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

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

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

The mode can be set independently in each environment.

Opening and Closing a Connection

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

The following code example creates an environment instance and then uses it to create a database connection for a database user scott with the password tiger.

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

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.

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

Connection Pooling

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

- Creating a Connection Pool
- Stateless Connection Pooling

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

Creating a Connection Pool

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

By employing the **connection pooling** feature, your application can create a small set of connections that can be used by a large number of threads. This enables you to use database resources very efficiently.

Creating a Connection Pool

To create a connection pool, you use 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;
```

The following parameters are used in the previous method example:

- `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

The following code example demonstrates how you can 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);
```

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 so as to maintain an optimum number of open connections.

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

Proxy Connections

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

A proxy connection can be created by using either of the following methods:

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

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.

Stateless Connection Pooling

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

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

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

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

Stateless connection pooling improves the scalability of the mid-tier applications by multiplexing the connections. However, connections from a

StatelessConnectionPool should not be used for long transactions, as holding connections for long periods leads to reduced concurrency.

Caution:

- OCCI will not check for the correctness of the connection-tag pair. The user is responsible for ensuring that connections with different client-side properties don't have the same tag.
- OCCI is not responsible for removing the state of the connection either by a commit or a rollback before releasing. If a state remains with a connection when it is released back to the pool, it will still be present when the connection is reused. The user is responsible for removing the state of the connection before releasing it back to the pool.

There are two types of stateless connection pools:

- A homogeneous pool is one in which all the connections will be authenticated with the username and password provided at the time of creation of the pool. Therefore, all connections will have the same authentication context. Proxy connections are not allowed in such pools.
- Different connections can be authenticated by different usernames. Proxy connections can also exist in such pools.

Example 2–1 presents the usage scenario for creating and using a homogeneous stateless connection pool, while Example 2–2 covers the use of heterogeneous pools.

Example 2–1 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 of the StatelessConnectionPool.

```
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 appropriately authenticated untagged connection with a `NULL` tag is returned.

Alternatively, you can obtain a connection with getAnyTaggedConnection() call. It will return 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, tag1);
   ```

   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 *conn1=scp->getConnection(tag1);
   ```

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

   ```
   scp->terminateConnection(conn1);
   ```

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

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

Example 2–2 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, 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 will return 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 wish to use proxy connections by getProxyConnection() or getAnyTaggedProxyConnection() calls on the `StatelessConnectionPool`.

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

3. Use the connection.

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

   ```
   scp->releaseConnection(conn, tag1);
   ```

   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 *conn1=scp->getConnection(tag1);
   ```

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

```
scp->terminateConnection(conn1);
```

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

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

Executing SQL DDL and DML Statements

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

Creating a Statement Object

To create a `Statement` object, call the `createStatement()` method of the `Connection` object, as shown in the following example:

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

Creating a Statement Object to Execute SQL Commands

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

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

Creating a Database Table

Using the `executeUpdate()` method, the following code example demonstrates how you can create a database table:

```
stmt->executeUpdate("CREATE TABLE basket_tab
   (fruit VARCHAR2(30), quantity NUMBER)");
```

Inserting Values into a Database Table

Similarly, you can execute a SQL `INSERT` statement by invoking the `executeUpdate()` method:

```
stmt->executeUpdate("INSERT INTO basket_tab
   VALUES('MANGOES', 3)");
```

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

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

Reusing a Statement Object

You can reuse a `Statement` object to execute SQL statements multiple times. For example, to repeatedly execute the same statement with different parameters, you specify the statement by the `setSQL` method of the `Statement` object:

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

You may now execute this `INSERT` statement as many times as required. If at a later time you wish to execute a different SQL statement, you simply reset the statement object. For example:

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

Thus, OCCI statement objects and their associated resources are not allocated or freed unnecessarily. You can retrieve the contents of the current statement object at any time by means of the `getSQL()` method.

Terminating a Statement Object

You should explicitly terminate and deallocate a Statement:

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

Types of SQL Statements in the OCCI Environment

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

- Standard Statements use SQL commands with specified values

- Parameterized Statements have parameters, or bind variables
- Callable Statements call stored PL/SQL procedures

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

Standard Statements

Previous sections describe examples of both DDL and DML commands. For example:

```
stmt->executeUpdate("CREATE TABLE basket_tab
   (fruit VARCHAR2(30), quantity NUMBER)");
```

and

```
stmt->executeUpdate("INSERT INTO basket_tab
   VALUES('MANGOES', 3)");
```

These are each an example of a **standard statement** in which you explicitly define the values of the statement. So, in these examples, the `CREATE TABLE` statement specifies the name of the table (`basket_tab`), and the INSERT statement stipulates the values to be inserted `('MANGOES', 3).`

Parameterized Statements

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

For example, suppose you want to execute an `INSERT` statement with different parameters. You first specify the statement by the `setSQL()` method of the `Statement` object:

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

You then call the `set`*`xxx`*`()` methods to specify the parameters, where *xxx* stands for the type of the parameter. The following example invokes the `setString()` and `setInt()` methods to input the values of these types into the first and second parameters.

To insert a row:

```
stmt->setString(1, "Bananas");     // value for first parameter
stmt->setInt(2, 5);                // value for second parameter
```

Having specified the parameters, you insert values into the row:

```
stmt->executeUpdate();             // execute statement
```

To insert another row:

```
stmt->setString(1, "Apples");      // value for first parameter
stmt->setInt(2, 9);                // value for second parameter
```

Having specified the parameters, you again insert values into the row:

```
stmt->executeUpdate();             // execute statement
```

If your application is executing the same statement repeatedly, then avoid changing the input parameter types because a rebind is performed each time the input type changes.

Callable Statements

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

For example, suppose you wish to call a procedure `countFruit()`, that returns the quantity of a specified kind of fruit. To specify the input parameters of a PL/SQL stored procedure, call the `set`*`XXX`*`()` methods of the `Statement` class as you would for parameterized statements.

```
stmt->setSQL("BEGIN countFruit(:1, :2); END:");
int quantity;
stmt->setString(1, "Apples");   // specify the first (IN) parameter of procedure
```

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

```
stmt->registerOutParam(2, Type::OCCIINT, sizeof(quantity));
   // specify the type and size of the second (OUT) parameter
```

You now execute the statement by calling the procedure:

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

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

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

Callable Statements with Arrays as 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,
   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`: Return code

Streamed Reads and Writes

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

Streamed data is of three kinds:

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

Methods of the Stream Class support the stream interface.

The getStream() method of the Statement Class returns a stream object that supports reading and writing for DML and callable statements:

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

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

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

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

To bind data in a streaming mode, follow these steps and review Example 2–3:

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 will be used in the streaming mode. If the bind position is a PL/SQL `IN` or `IN/OUT` argument type, indicate this by calling the three-argument versions of these methods and setting the `inArg` parameter to `TRUE`.
3. Execute the statement; the status() method of the Statement Class will return `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 will change to an appropriate value, such as `UPDATE_COUNT_AVAILABLE`.

Example 2–3 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, 100000, TRUE);
```

Fetching Data in a Streaming Mode: PL/SQL

To fetch data from a streaming mode, follow these steps and review Example 2–4:

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 will be retrieved from the streaming mode.
3. Execute the statement; the status() method of the Statement Class will return `STREAM_DATA_AVAILABLE`.
4. Obtain the stream object through a getStream() method of the Statement Class.
5. Use readBuffer() and readLastBuffer() methods of the Stream Class to read data.
6. Close the stream with closeStream() method of the Statement Class.

Example 2–4 How to Fetch Data in a Streaming Mode Using PL/SQL

```
Statement *stmt = conn->createStatement("BEGIN testproc(:1); END;");
                           //argument 1 is OUT type
stmt->setCharacterStreamMode(1, 100000);
stmt->execute();

Stream *outarg = stmt->getStream(1);
                           //use Stream::readBuffer/readLastBuffer to read data
```

Fetching Data in Streaming Mode: ResultSet

Executing SQL Queries and Example 2–6 on page 2-21 provide an explanation of how to use the streaming interface with result sets.

Working with Multiple Streams

If you have to work with multiple read and write streams, you have to ensure that the read or write of one stream is completed prior to reading or writing on another stream. To determine stream position, use the getCurrentStreamParam() method of the Statement Class or ResultSet Class. Example 2–5 illustrates how to work with concurrent streams.

Example 2–5 How to Work with Multiple Streams

```
Statement *stmt = conn->createStatement(
   "Insert into testtab(longcol1, longcal2) values (:1,:2)");
                                           //longcol1 AND longcol2 are 2 columns
                                           //inserted in streaming mode
stmt->setBinaryStreamMode(1, 100000);
stmt->setBinaryStreamMode(2, 100000);
stmt->executeUpdate();
```

```
Stream *col1 = stmt->getStream(1);
Stream *col2 = stmt->getStream(2);

col1->writeBuffer(buffer, len);          //first stream
...                                      //complete writing col1 stream
col1->writeLastBuffer(buffer, len);      //first and then move to col2

col2->writeBuffer(buffer, len);          //second stream
...
```

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. To do this, use the `addIteration()` method of the `Statement` class to perform batch operations that modify a different row with each iteration.

To execute `INSERT`, `UPDATE`, and `DELETE` operations iteratively, you must:

- Set the maximum number of iterations
- Set the maximum parameter size for variable length parameters

Setting the Maximum Number of Iterations

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

```
Statement->setMaxIterations(int maxIterations);
```

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

Setting the Maximum Parameter Size

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

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

You do not need to set the maximum parameter size for fixed length datatypes, such as `Number` and `Date`, or for parameters that use the `setDataBuffer()` method.

You can retrieve the current maximum parameter size setting by calling the `getMaxParamSize()` method.

Executing an Iterative Operation

Once you have set the maximum number of iterations and (if necessary) the maximum parameter size, iterative execution using a parameterized statement is straightforward, as shown in the following example:

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

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

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

stmt->executeUpdate();             // execute statement
```

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

Iterative Execution Usage Notes

- 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.
- The datatype 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.

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 as a result of a query.

Result Set

Execution of a database query puts the results of the query into a set of rows called the result set. In OCCI, a SQL `SELECT` statement is executed by the `executeQuery` method of the `Statement` class. This method returns 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 an enumerated type of `Status`. The possible values of `Status` are:

- `DATA_AVAILABLE`
- `END_OF_FETCH = 0`
- `STREAM_DATA_AVAILABLE`

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 stream data is successfully read, as shown in the following code example:

```
ResultSet *rs = stmt->executeQuery("SELECT * FROM demo_tab");
ResultSet::Status status = rs->status();     // status is DATA_AVAILABLE
while (rs->next())
{
   get data and process;
}
```

When the entire result set has been traversed, then the status changes to END_OF_FETCH which terminates the WHILE loop.

Example 2–6 illustrates the previously described steps.

Example 2–6 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;
}
```

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;

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

Optimizing Performance by Setting Prefetch Count

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

You effect this either by setting the number of rows to be prefetched through the `setPrefetchRowCount()` method, or by setting the memory size to be used for prefetching through the `setPrefetchMemorySize()` method.

If you set both of these attributes, then the specified number of rows are prefetched unless the specified memory limit is reached first. If the specified memory limit is reached first, then the prefetch returns as many rows as will 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`.

Note: Prefetching is not in effect if LONG columns are part of the query. Queries containing LOB columns *can* be prefetched, because the LOB locator, rather than the data, is returned by the query.

Executing Statements Dynamically

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

If your application needs to allow for dynamic events and you cannot be sure of which statement will need to 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 will return one of these statuses, you can also 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
```

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
```

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

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
```

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 will also change the status to `PREPARED`. For example:

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

RESULT_SET_AVAILABLE

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

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

```
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();
```

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:

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

Note that a DDL statement will result in an update count of zero (0). Similarly, an update that does not meet any matching conditions will also produce a count of zero (0). In such a case, you cannot infer the kind of statement that has been executed from the reported status.

NEEDS_STREAM_DATA

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

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

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 needs to be read.

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

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

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

Committing a Transaction

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

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

```
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);
```

Statement Caching

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 2–7 and Example 2–8 for typical usage scenarios.

Example 2–7   Statement Caching without Connection Pooling

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

1. Create a `Connection` by making a createConnection() call on the `Environment` object.

   ```
   Connection *conn = env->createConnection(username, password, connecstr);
   ```

2. Enable statement caching on the `Connection` object by using a nonzero `size` parameter in the setStmtCacheSize() call.

   ```
   conn->setStmtCacheSize(10);
   ```

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

3. Create a `Statement` by making a createStatement() call on the `Connection` object; the `Statement` is returned if it is in the cache already, 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 don't want to cache this statement, use the disableCaching() call and an alternate from of terminateStatement():

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

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

6. Terminate the connection.

Example 2–8   Statement Caching with Connection Pooling

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

1. Create a `ConnectionPool` by making a call to the createStatelessConnectionPool() 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);
   ```

   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 already, 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 already, 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 don't want to cache this statement, use the disableCaching() call and an alternate from of terminateStatement():

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

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

7. Release the connection terminateConnection().

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

Note:

- 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.

See Also:

- Connection Class methods: createStatement() on page 10-63, getStmtCacheSize() on page 10-67, isCached() on page 10-67, setStmtCacheSize() on page 10-70, and terminateStatement() on page 10-71.
- ConnectionPool Class methods: getStmtCacheSize()getStmtCacheSize() on page 10-76 and terminateConnection() on page 10-77.
- StatelessConnectionPool Class methods: getStmtCacheSize(), setStmtCacheSize() on page 10-258and terminateConnection() on page 10-259.
- Statement Class method disableCaching() on page 10-266.

Exception Handling

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

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

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

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

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

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

```
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: *Oracle Database Error Messages* for more information about Oracle error messages.

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 to not raise an `SQLException`. In this case, null data is returned as zero (`0`) for numeric values and null strings for character values.

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 2–1, Table 2–2, and Table 2–3.

Table 2–1 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 2–2 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 2–3 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 2–3, 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`.

Advanced Relational Techniques

The following advanced techniques are discussed in this section:

- Sharing Connections
- Optimizing Performance

Sharing Connections

This section covers the following topics:

- Overview ofThread Safety
- Thread Safety and Three-Tier Architectures
- Implementing Thread Safety
- Serialization

Overview ofThread 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, machine registers, and stack. Global and static variables are common to all threads, and a mutual exclusivity mechanism may be required to manage access to these variables from multiple threads within an application.

Once spawned, threads run asynchronously to one another. They can access common data elements and make OCCI calls in any order. Because of this shared access to data elements, a mechanism is required to maintain the integrity of data being accessed by multiple threads. The mechanism to manage data access takes the form of mutexes (mutual exclusivity locks), which ensure that no conflicts arise between multiple threads that are accessing shared resources within an application. In OCCI, mutexes are granted on an OCCI environment basis.

This thread safety feature of the Oracle database server and OCCI library enables developers to use OCCI in a multithreaded application with these added benefits:

- Multiple threads of execution can make OCCI calls with the same result as successive calls made by a single thread.
- When multiple threads make OCCI calls, there are no side effects between threads.
- Even if you do not write a multithreaded program, you do not pay any performance penalty for including thread-safe OCCI calls.
- Use of multiple threads can improve program performance. You can discern gains on multiprocessor systems where threads run concurrently on separate processors, and on single processor systems where overlap can occur between slower operations and faster operations.

Thread Safety and Three-Tier Architectures

In addition to client/server applications, where the client can be a multithreaded program, thread safety is typically used in three-tier or client/agent/server architectures. In this architecture, the client is concerned only with presentation services. The agent (or application server) processes the application logic for the client application. Typically, this relationship is a many-to-one relationship, with multiple clients sharing the same application server.

The server tier in the three-tier architecture is an Oracle database server. The applications server (agent) supports multithreading, with each thread serving a separate client application. In an Oracle environment, this middle-tier application server is an OCCI or precompiler program.

Implementing Thread Safety

In order 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.

If a multithreaded application is running in a thread-safe operating system, then the OCCI library will manage 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.

Note:

- Applications running 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, whether or not 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.

Serialization

As an application programmer, you have two basic options regarding concurrency in a multithreaded application:

- Automatic serialization, in which you utilize OTIS's transparent mechanisms
- Application-provided serialization, in which you manage the contingencies involved in maintaining multiple threads

Automatic Serialization In cases where there are multiple threads operating on objects (connections and connection pools) derived from an OCCI environment, you can elect to let OCCI serialize access to those objects. The first step is to pass a value of `THREADED_MUTEXED` to the `createEnvironment` method. At this point, the OCCI library automatically acquires a mutex on thread-safe objects in the environment.

When the OCCI environment is created with `THREADED_MUTEXED` mode, then only the `Environment`, `Map`, `ConnectionPool`, `StatelessConnectionPool` and `Connection` objects are thread-safe. That is, if two threads make simultaneous calls on one of these objects, then OCCI serializes them internally. However, note that all other OCCI objects, such as `Statement`, `ResultSet`, `SQLException`, `Stream`, and so on, are not thread-safe as, applications should not operate on these objects simultaneously from multiple threads.

Note that the bulk of processing for an OCCI call happens on the server, so if two threads that use OCCI calls go to the same connection, then one of them could be blocked while the other finishes processing at the server.

Application-Provided Serialization In cases where there are multiple threads operating on objects derived from an OCCI environment, you can chose to manage serialization. The first step is to pass a value of `THREADED_UNMUTEXED` for the `createEnvironment` mode. In this case the application must mutual exclusively lock OCCI calls made on objects derived from the same OCCI environment. This has the advantage that the mutex scheme can be optimized based on the application design to gain greater concurrency.

When an OCCI environment is created in this mode, OCCI recognizes that the application is running in a multithreaded application, but that OCCI need not acquire its internal mutexes. OCCI assumes that all calls to methods of objects derived from that OCCI environment are serialized by the application. You can achieve this two different ways:

- Each thread has its own environment. That is, the environment and all objects derived from it (connections, connection pools, statements, result sets, and so on) are not shared across threads. In this case your application need not apply any mutexes.
- If the application shares an OCCI environment or any object derived from the environment across threads, then it must serialize access to those objects (by using a mutex, and so on) such that only one thread is calling an OCCI method on any of those objects.

Basically, 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.

Note:

- 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 with regard to connections, connection pools, statements, and result set objects. This will result in further memory consumption.
- Having multiple connections to the server results in more resource consumptions on the server and network. Having multiple environments would normally entail more connections.

Optimizing Performance

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 executeArrayUpdate() method of the Statement Class.

setDataBuffer() Method

For high performance applications, OCCI provides the `setDataBuffer` method whereby the data buffer is managed by the application. The following example shows the setDataBuffer() method:

```
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 datatypes can be provided and retrieved by means of the `setDataBuffer()` method. For instance, C++ Standard Library strings cannot be provided with the `setDataBuffer()` interface.

> **See Also:** Table 4–2, " External Datatypes and Corresponding C++ and OCCI Types" in Chapter 4, "Datatypes" 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` prior to execute. The value of `str1` that is used is the value at the time `setString(str1)` is called. However, for data provided by means of 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.

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 an example that inserts two employees in the `emp` table:

```
Statement *stmt = conn->createStatement("insert into emp (id, ename)
   values(:1, :2)");
char enames[2][] = {"SMITH", "MARTIN"};
ub2 enameLen[2];
for (int i = 0; i < 2; i++)
   enameLen[i] = strlen(enames[i] + 1);
stmt->setMaxIteration(2);             // set maximum number of iterations
stmt->setInt(1, 7369);                // specify data for the first row
stmt->setDataBuffer(2, enames, OCCI_SQLT_STR, sizeof(ename[0]), &enameLen);
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();
```

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:

```
stmt ->setSQL("insert into emp (id, ename) values (:1, :2)");
char enames[2][] = {"SMITH", "MARTIN"};
ub2 enameLen[2];
for (int i = 0; i < 2; i++)
   enameLen[i] = strlen(enames[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, enames, OCCI_SQLT_STR, sizeof(ename[0]), &len);
stmt->executeArrayUpdate(2);          // data for two rows is inserted.
```

Array Fetch Using next() Method

If the application is fetching data with only the `setDataBuffer()` interface or the stream interface, then an array fetch can be executed. The array fetch is implemented by calling the `ResultSet->next(int numRows)` method. This causes up to `numRows` amount of data to be fetched for each column. The buffers specified with the `setDataBuffer()` interface should be big enough to hold data for multiple rows. Data for the ith row is fetched at `buffer + (i - 1) * size` location. Similarly, the length of the data is stored at `*(length + (i - 1))`.

```
int empno[5];
char ename[5][11];
ub2  enameLen[5];
ResultSet *resultSet = stmt->executeQuery("select empno, ename  from emp");
resultSet->setDataBuffer(1, &empno, OCCIINT);
resultSet->setDataBuffer(2, ename, OCCI_SQLT_STR, sizeof(ename[0]), enameLen);
rs->next(5);           // fetches five rows, enameLen[i] has length of ename[i]
```

3

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

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 object 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 OCCI's API for navigational access, your application can perform the following functions on Oracle objects:

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

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

Working with Objects in OCCI

Many of the programming principles that govern a relational OCCI applications are identical for object-relational applications. An object-relational application uses the standard OCCI calls to establish database connections and process SQL statements. The difference is that the SQL statements that are issued retrieve object references, which can then be manipulated with OCCI's 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 or not they are referenced by way of an object identifier.

Persistent Objects

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

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

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

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

Note:

- Users don't 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 following SQL examples demonstrate the difference between these two types of persistent objects.

Example 3–1 Creating Standalone Objects

This code example demonstrates how a standalone object is created:

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

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.

Example 3–2 Creating Embedded Objects

This code example demonstrates how an embedded object is created:

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

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. This means they cannot be pinned in an OCCI application, and they also never need to be unpinned. They are always retrieved into the object cache *by value*.

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

Transient Objects

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

The Object Type Translator (OTT) utility generates two `operator new` methods for each C++ class, as demonstrated in this code example:

```
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
}
```

The following code example demonstrates how a transient object can be created:

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

Transient objects cannot be converted to persistent objects. Their role is fixed at the time they are instantiated. It is also the user's responsibility to free memory by deleting transient objects.

See Also:

- *Oracle Database Concepts* for more information about objects

Values

In the context of this manual, a **value** refers to either:

- A scalar value which is stored in a nonobject 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.

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

Representing Objects in C++ Applications

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

Creating Persistent and Transient Objects

The following sections discuss how persistent and transient objects are created.

Example 3–3 Creating a Persistent Object

Before you create a persistent object, you must have created the environment and opened a connection. The following example shows how to create a persistent object, `addr`, in the database table, `addr_tab`, created by means of a SQL statement:

```
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");
```

The 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 3–4 Creating a Transient Object

An instance of the transient object `ADDRESS` is created in the following manner:

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

Creating Object Representations using the OTT Utility

When your C++ application retrieves instances of object types from the database, it needs to have a client-side representation of the objects. The Object Type Translator (OTT) utility generates C++ class representations of database object types for you. For example, consider the following declaration of a type in your 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 Session* sess,
         const string& table);
      string  getSQLTypeName() const;
      void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
                          unsigned int &schemaNameLen, void **typeName,
                          unsigned int &typeNameLen) const;
      ADDRESS(void *ctx) : PObject(ctx) { };
      static void *readSQL(void *ctx);
      virtual void readSQL(AnyData& stream);
      static void writeSQL(void *obj, void *ctx);
      virtual void writeSQL(AnyData& stream);
}
```

These class declarations are automatically written by OTT to a header (`.h`) 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` (as well as 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: Chapter 6, "Object Type Translator Utility" for more information about the OTT utility

Developing an OCCI Object Application

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

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 will most likely need to include C++ class representations of database objects in a header file. You can create these classes by using the Object Type Translator (OTT) utility, as described in Chapter 6, "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.

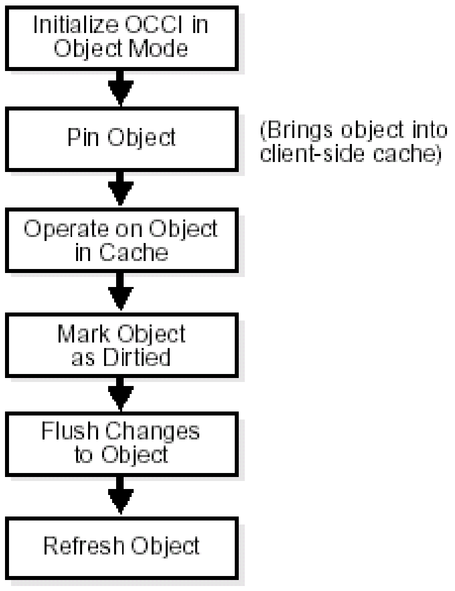
See Also:

- Chapter 2, "Relational Programming" for information about using OCCI to connect to a database server, process SQL statements, and allocate handles
- Chapter 6, "Object Type Translator Utility" for information about the OTT utility
- Chapter 10, "OCCI Application Programming Interface" for descriptions of OCCI relational functions and the `Connect` class and the `getMetaData` method

Basic Object Operational Flow

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

Figure 3–1 Basic Object Operational Flow


Initialize OCCI in Object Mode
Pin Object
(Brings object into client-side cache)
Operate on Object in Cache
Mark Object as Dirtied
Flush Changes to Object
Refresh Object

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

Initialize OCCI in Object Mode

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

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

Note: If the `mode` parameter is not set to `OBJECT`, any attempt to use an object-related function will result 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;

env = Environment::createEnvironment(Environment::OBJECT);
con = env->createConnection(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 the case of 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 will continue to be pinned in the object cache and be accessible by the OCCI client.

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

Pin Object

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

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

Operate on Object in Cache

Note that the object cache does not manage the contents of object copies; it does not automatically refresh object copies. Your application must ensure the validity and consistency of object copies.

Flush Changes to Object

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

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

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

Note: If you initialize the OCCI environment in object mode, your application allocates memory for the object cache, whether or not the application actually uses object calls.

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

Deletion of an Object

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

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

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

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

Migrating C++ Applications Using OCCI

This section will describe how to migrate existing C++ applications using OCCI.

Steps for Migration

- Determine object model and class hierarchy
- Use JDeveloper9i to map to Oracle object schema
- Generate C++ header files using Oracle Type Translator
- Modify old C++ access classes as required to work with new object type definitions
- Add functionality for transient and persistent object management, as required.

Overview of Associative Access

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

- Using SQL to Access Objects
- Inserting and Modifying Values

See Also: complete code listing of the demonstration programs

Using SQL to Access Objects

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

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();
```

Overview of Navigational Access

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

- Retrieving an Object Reference (REF) from the Database Server
- Pinning an Object
- Manipulating Object Attributes
- Marking Objects and Flushing Changes

See Also: complete code listing of the demonstration programs

Retrieving an Object Reference (REF) from the Database Server

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

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

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

```
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: "Executing SQL DDL and DML Statements" on page 2-10 for general information about preparing and executing SQL statements

Pinning an Object

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.

Note: This section deals with a simple pin operation involving a single object at a time. For information about retrieving multiple objects through complex object retrieval, see the section Overview of Complex Object Retrieval on page 3-18.

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.

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 not already in the object cache.

Marking Objects and Flushing Changes

In the example in the previous section, an attribute of an object was changed. At this point, however, that change exists only in the client-side cache. The application must take specific steps to ensure that the change is written to the database.

Marking an Object as Modified (Dirty)

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

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

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

Recording Changes in the Database

Objects that have had their dirty flag set must be flushed to the database server for the changes to be recorded in the database. This can be done in three ways:

- Flush a single object marked dirty by calling the method `flush`, a derived method of `PObject`.

- Flush the entire object cache using the `Connection::flushCache()` method. In this case, 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.

Garbage Collection 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 to 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: 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;`

"Pin Object" on page 3-11 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. In the case of a newly created persistent object, the object is unmarked after the transaction is committed or aborted and if the object has a `0` pin count, in other words there are no references referring to it. The object then becomes a candidate for being aged 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:

- Chapter 10, "OCCI Application Programming Interface" for details.

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>` is the same as any C++ pointer and it provides access to the object copy in the cache. But once 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.

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 through the use of **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.

Note: 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 need 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.

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

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 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 will very likely need access to the customer information for that purchase order. Using simple navigation, this would require two database server accesses to retrieve the two objects.

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

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

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

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

```
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);
```

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 already 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 already pinned leading to a performance degradation instead of performance improvement.

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

The `SELECT` privilege is needed for all prefetched objects. Objects in the complex object for which the application does not have `SELECT` privilege will not be 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.

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. The following is the SQL DDL to create a VARRAY and an object that contains an attribute of type VARRAY.

```
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 Session* sess,
      const string& table);
      string  getSQLTypeName() const;
      void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
                          unsigned int &schemaNameLen, void **typeName,
                          unsigned int &typeNameLen) const;
      PERSON (void *ctx) : PObject(ctx) { };
      static void *readSQL(void *ctx);
      virtual void readSQL(AnyData& stream);
      static void writeSQL(void *obj, void *ctx);
      virtual void writeSQL(AnyData& stream);
}
```

See Also: complete code listing of the demonstration programs

Fetching Embedded Objects

If your application needs to fetch an embedded object — 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. This means that they cannot serve as the basis for object navigation. There are still many situations, however, in which an application will want to fetch 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));
```

You could then use that type as the datatype 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" on page 3-12.

Nullness

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

- Any attribute of an object can have a NULL value. This indicates that the value of that attribute of the object is not known.

- An object may be **atomically NULL**. This means that the value of the entire object is unknown.

Atomic NULLness is not the same thing as nonexistence. An atomically NULL object still exists, its value is just not known. It may be thought of as an existing object with no data.

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

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

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

Using Object References

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

Deleting Objects from the Database

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

Type Inheritance

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

> **See Also:** *Oracle Database Application Developer's Guide - Object-Relational Features* 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 release 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:**
>
> - "OTT Support for Type Inheritance" on page 3-27 for the classes generated by OTT for this example.

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

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

Substitutability

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

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

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

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

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

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

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

```
Book_t('My Oracle Experience',
   Employee_t(12345, 'Joe', 'SF', 1111, NULL))
```

NOT INSTANTIABLE Types and Methods

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

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

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

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

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

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

OCCI Support for Type Inheritance

The following calls support type inheritance.

Connection::getMetaData()

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

Bind and Define Functions

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

OTT Support for Type Inheritance

Class declarations for objects with inheritance are similar to the simple object declarations except that the class is derived from the parent type class and only the fields corresponding to attributes not already in the parent class are included. The structure for these declarations is listed in Example 3–5:

Example 3–5 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 Session* sess, const string& table);
      string  getSQLTypeName() const;
      void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
                          unsigned int &schemaNameLen, void **typeName,
                          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);
}
```

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.

A Sample OCCI Application

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

Example 3–6 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 3–7   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 3–8   Listing of OTT Command that Generates Files for a Sample OCCI Application

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

Example 3–9   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 3–10 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("SCOTT.FULLNAME", &CFullName::readSQL, &CFullName::writeSQL);
  mapOCCI_->put("SCOTT.ADDRESS", &CAddress::readSQL, &CAddress::writeSQL);
  mapOCCI_->put("SCOTT.PERSON", &CPerson::readSQL, &CPerson::writeSQL);
  mapOCCI_->put("SCOTT.STUDENT", &CStudent::readSQL, &CStudent::writeSQL);
}
```

Example 3–11 Listing of demo2.h for a Sample OCCI Application

```
#ifndef DEMO2_ORACLE
# define DEMO2_ORACLE

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

/* Make the foll changes to the generated file */
using namespace std;
using namespace oracle::occi;

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

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

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

public:
   OCCI_STD_NAMESPACE::string getFirst_name() const;
   void setFirst_name(const OCCI_STD_NAMESPACE::string &value);
   OCCI_STD_NAMESPACE::string getLast_name() const;
   void setLast_name(const OCCI_STD_NAMESPACE::string &value);
   void *operator new(size_t size);
   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);
   void *operator new(size_t, 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;

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,
      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 3–12   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)
{
  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 *) "SCOTT.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("SCOTT.FULLNAME");
}

void CFullName::getSQLTypeName(oracle::occi::Environment *env,
     void  **schemaName, unsigned int &schemaNameLen, void **typeName,
     unsigned int &typeNameLen) const
{
  PObject::getSQLTypeName(env, &CFullName::readSQL, schemaName,
        schemaNameLen, typeName, typeNameLen);
}

CFullName::CFullName()
{
}

void *CFullName::readSQL(void *ctxOCCI_)
{
  MyFullName *objOCCI_ = new(ctxOCCI_) MyFullName(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

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

```
void CFullName::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   FIRST_NAME = streamOCCI_.getString();
   LAST_NAME = streamOCCI_.getString();
}

void CFullName::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CFullName *objOCCI_ = (CFullName *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void CFullName::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setString(FIRST_NAME);
   streamOCCI_.setString(LAST_NAME);
}

CFullName::~CFullName()
{
  int i;
}

/* GENERATED METHOD IMPLEMENTATIONS FOR THE ADDRESS OBJECT TYPE. */
OCCI_STD_NAMESPACE::string CAddress::getState() const
{
  return STATE;
}

void CAddress::setState(const OCCI_STD_NAMESPACE::string &value)
{
  STATE = value;
}

OCCI_STD_NAMESPACE::string CAddress::getZip() const
{
  return ZIP;
}

void CAddress::setZip(const OCCI_STD_NAMESPACE::string &value)
{
  ZIP = value;
}

void *CAddress::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *CAddress::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.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("SCOTT.ADDRESS");
}
```

```
void CAddress::getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
  unsigned int &schemaNameLen, void **typeName, unsigned int &typeNameLen) const
{
  PObject::getSQLTypeName(env, &CAddress::readSQL, schemaName,
        schemaNameLen, typeName, typeNameLen);
}

CAddress::CAddress()
{
}

void *CAddress::readSQL(void *ctxOCCI_)
{
  MyAddress *objOCCI_ = new(ctxOCCI_) MyAddress(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

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

void CAddress::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   STATE = streamOCCI_.getString();
   ZIP = streamOCCI_.getString();
}

void CAddress::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CAddress *objOCCI_ = (CAddress *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void CAddress::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setString(STATE);
   streamOCCI_.setString(ZIP);
}

CAddress::~CAddress()
{
  int i;
}

/* GENERATED METHOD IMPLEMENTATIONS FOR THE PERSON OBJECT TYPE. */
oracle::occi::Number CPerson::getId() const
{
  return ID;
}

void CPerson::setId(const oracle::occi::Number &value)
{
  ID = value;
}

MyFullName * CPerson::getName() const
{
  return NAME;
}

void CPerson::setName(MyFullName * value)
{
  NAME = value;
}

oracle::occi::Ref< MyAddress > CPerson::getCurr_addr() const
```

```
{
  return CURR_ADDR;
}

void CPerson::setCurr_addr(const oracle::occi::Ref< MyAddress > &value)
{
  CURR_ADDR = value;
}

OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > >& CPerson::getPrev_
addr_l()
{
  return PREV_ADDR_L;
}

const OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > >&
  CPerson::getPrev_addr_l() const
{
  return PREV_ADDR_L;
}

void CPerson::setPrev_addr_l(const OCCI_STD_NAMESPACE::vector<
 oracle::occi::Ref< MyAddress > > &value)
{
  PREV_ADDR_L = value;
}
void *CPerson::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *CPerson::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.PERSON");
}

void *CPerson::operator new(size_t size, void *ctxOCCI_)
{
 return oracle::occi::PObject::operator new(size, ctxOCCI_);
}

void *CPerson::operator new(size_t size,
    const oracle::occi::Connection *sess,
    const OCCI_STD_NAMESPACE::string &tableName,
    const OCCI_STD_NAMESPACE::string &typeName,
    const OCCI_STD_NAMESPACE::string &tableSchema,
    const OCCI_STD_NAMESPACE::string &typeSchema)
{
  return oracle::occi::PObject::operator new(size, sess, tableName,
        typeName, tableSchema, typeSchema);
}

OCCI_STD_NAMESPACE::string CPerson::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.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;
}

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 *) "SCOTT.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("SCOTT.STUDENT");
}

void CStudent::getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
  unsigned int &schemaNameLen, void **typeName, unsigned int &typeNameLen) const
{
  PObject::getSQLTypeName(env, &CStudent::readSQL, schemaName,
        schemaNameLen, typeName, typeNameLen);
```

```
}

CStudent::CStudent()
{
}
void *CStudent::readSQL(void *ctxOCCI_)
{
  MyStudent *objOCCI_ = new(ctxOCCI_) MyStudent(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      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);
}

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

Let us assume OTT generates `FULL_NAME`, `ADDRESS`, `PERSON`, and `PFGRFDENT` class declarations in `demo2.h`. The following sample OCCI application will extend the classes generated by OTT, as specified in `demo2.typ` file in Example 3–7, and will add some user defined methods. Note that these class declaration have been incorporated into `demo2.h` to ensure correct compilation.

Example 3–13 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 3–14 Listing for myDemo.cpp for a Sample OCCI Application

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

/* initialize MyFullName */
MyFullName::MyFullName(string first_name, string last_name)
{
 setFirst_name(first_name);
 setLast_name(last_name);
}

/* display all the information in MyFullName */
void MyFullName::displayInfo()
{
   cout << "FIRST NAME is" << getFirst_name() << endl;
   cout << "LAST NAME is" << getLast_name() << endl;
}

MyFullName::MyFullName(void *ctxOCCI_):CFullName(ctxOCCI_)
{
}

/* METHOD IMPLEMENTATIONS FOR MyAddress CLASS. */

/* initialize MyAddress */
MyAddress::MyAddress(string state_i, string zip_i)
{
  setState(state_i);
  setZip(zip_i);
}

/* display all the information in MyAddress */
void MyAddress::displayInfo()
{
   cout << "STATE is" << getState() << endl;
   cout << "ZIP is" << getZip() << endl;
}

MyAddress::MyAddress(void *ctxOCCI_) :CAddress(ctxOCCI_)
{
}

/* METHOD IMPLEMENTATIONS FOR MyPerson CLASS. */

/* initialize MyPerson */
MyPerson::MyPerson(Number id_i, MyFullName* name_i,
                   const Ref<MyAddress>& addr_i)
{
  setId(id_i);
  setName(name_i);
  setCurr_addr(addr_i);
}

MyPerson::MyPerson(void *ctxOCCI_) :CPerson(ctxOCCI_)
{
}

/* move Person from curr_addr to new_addr */
void MyPerson::move(const Ref<MyAddress>& new_addr)
{
   // append curr_addr to the vector //
   getPrev_addr_l().push_back(getCurr_addr());
   setCurr_addr(new_addr);

   // mark the object as dirty
   this->markModified();

/*  display all the information of MyPerson */
void MyPerson::displayInfo()
{
   cout << "ID is" << (int)getId() << endl;
   getName()->displayInfo();
   // de-referencing the Ref attribute using -> operator
   getCurr_addr()->displayInfo();
   cout << "Prev Addr List: " << endl;
   for (int i = 0; i < getPrev_addr_l().size(); i++)
   {
      // access the collection elements using [] operator
      (getPrev_addr_l())[i]->displayInfo();
   }
}

MyPerson::MyPerson()
{
}

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

Example 3–15 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;

int main()
{
   Environment *env = Environment::createEnvironment(Environment::OBJECT);
   mappings(env);

   try {
     Connec tion *conn = env->createConnection("SCOTT", "TIGER");

    /* Call the OTT generated function to register the mappings */
    /* create a persistent object of type ADDRESS in the database table,
       ADDR_TAB */
    MyAddress *addr1 = new(conn, "ADDR_TAB") MyAddress("CA", "94065");
```

```
     conn->commit();

     Statement *st = conn->createStatement("select ref(a) from addr_tab a");
    ResultSet *rs = st->executeQuery();
    Ref<MyAddress> r1;
    if ( rs->next())
       r1 = rs->getRef(1);
    st->closeResultSet(rs);
    conn->terminateStatement(st);

    MyFullName * name1 = new MyFullName("Joe", "Black");

    /* create a persistent object of type Person in the database table
       PERSON_TAB */
    MyPerson *person1 = new(conn, "PERSON_TAB") MyPerson(1,name1,r1);
    conn->commit();

    /* selecting the inserted information */
    Statement *stmt = conn->createStatement();
    ResultSet *resultSet =
         stmt->executeQuery("SELECT REF(a) from person_tab a where id = 1");

    if (resultSet->next())
    {
       Ref<MyPerson> joe_ref = (Ref<MyPerson>) resultSet->getRef(1);
       joe_ref->displayInfo();

       /* create a persistent object of type ADDRESS in the database table
          ADDR_TAB */
       MyAddress *new_addr1 = new(conn, "ADDR_TAB") MyAddress("PA", "92140");
       joe_ref->move(new_addr1->getRef());
       joe_ref->displayInfo();
    }

    /* commit the transaction which results in the newly created object
        new_addr and the dirty object joe to be flushed to the server.
        Note that joe was marked dirty in move(). */
    conn->commit();

    conn->terminateStatement(stmt);
    env->terminateConnection(conn);
  }

  catch ( exception &x)
```

```
 {
  cout << x.what () << endl;
 }
   Environment::terminateEnvironment(env);
   return 0;
}
```

4 Datatypes

This chapter is a reference for Oracle datatypes used by Oracle C++ Interface applications. This information will help you 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 Datatypes

Accurate communication between your C++ program and the Oracle database server is critical. OCCI applications can retrieve data from database tables by using SQL queries or they can modify existing data through the use of 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++ datatypes are converted to Oracle datatypes and back again.

In the Oracle database, values are stored in columns in tables. Internally, Oracle represents data in particular formats called internal datatypes. `NUMBER`, `VARCHAR2`, and `DATE` are examples of Oracle internal datatypes.

OCCI applications work with host language datatypes, or external datatypes, 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 datatypes to external datatypes.

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 datatypes. When data is sent to the database server from the OCCI application, the external datatype 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 datatype 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 datatype is explicitly specified in the method call). Similarly, when data values are fetched through a `ResultSet` object, the external representation of the retrieved data must be specified. This is done by calling a `get`*`xxx`*`()` method (the `external datatype` is implicitly specified by the method name) or by calling the `setDataBuffer()` method (the external datatype is explicitly specified in the method call).

Note: There are more external datatypes than internal datatypes. In some cases, a single external datatype maps to a single internal datatype; in other cases, many external datatypes map to a single internal datatype. The many-to-one mapping provides you with added flexibility.

See Also: External Datatypes on page 4-5

Internal Datatypes

The internal (built-in) datatypes provided by Oracle are listed in this section. A brief summary of internal Oracle datatypes, including description, code, and maximum size, appears in Table 4–1.

Table 4–1 Summary of Oracle Internal Datatypes

Internal Datatype	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`	4,000 bytes
`RAW`	2,000 bytes
`REF`	
`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)	
`VARCHAR2`	4,000 bytes

See Also:

- *Oracle Database SQL Reference*
- *Oracle Database Concepts*

Character Strings and Byte Arrays

You can use five Oracle internal datatypes 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 machines by using different languages (where single characters may be represented by differing numbers of bytes) can be significantly changed in length. Raw data is never converted in this way.

The database designer is responsible for choosing the appropriate Oracle internal datatype for each column in a table. You must be aware of the many possible ways that character and byte-array data can be represented and converted between variables in the OCCI program and Oracle database tables.

Universal Rowid (UROWID)

The universal rowid (UROWID) is a datatype 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 datatype, 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)

External Datatypes

Communication between the host OCCI application and the Oracle database server is through the use of external datatypes. Specifically, external datatypes are mapped to C++ datatypes.

Table 4–2 lists the Oracle external datatypes, the C++ equivalent (what the Oracle internal datatype is usually converted to), and the corresponding OCCI type. In C++ Datatype column, n stands for variable length and depends on program requirements or operating system. External datatypes marked with an asterix, (*), are used for setDataBuffer() and setDataBufferArray() interfaces, while methods marked with a double asterix, (**), are used for registerOutParam() interface.

Table 4–2 External Datatypes and Corresponding C++ and OCCI Types

External Datatype	C++ Type	OCCI Type
16 bit signed INTEGER *	signed short, signed int	OCCIINT
32 bit signed INTEGER *	signed int, signed long	OCCIINT
8 bit signed INTEGER *	signed char	OCCIINT
BFILE **	Bfile	OCCIBFILE
Binary FILE *	LNOCILobLocator	OCCI_SQLT_FILE
Binary LOB *	LNOCILobLocator	OCCI_SQLT_BLOB
BLOB **	Blob	OCCIBLOB
BOOL **	bool	OCCIBOOL
BYTES **	Bytes	OCCIBYTES
CHAR *	char[n]	OCCI_SQLT_AFC
CHAR **	string	OCCICHAR
Character LOB *	LNOCILobLocator	OCCI_SQLT_CLOB
CHARZ *	char[n+1]	OCCI_SQLT_RDD
CLOB **	Clob	OCCICLOB
CURSOR **	ResultSet	OCCICURSOR

Table 4–2 (Cont.) External Datatypes and Corresponding C++ and OCCI Types

External Datatype	C++ Type	OCCI Type
DATE *	char[7]	OCCI_SQLT_DAT
DATE **	Date	OCCIDATE
DOUBLE **	double	OCCIDOUBLE
FLOAT *	float, double	OCCIFLOAT
FLOAT **	float	OCCIFLOAT
INT **	int	OCCIINT
INTERVAL DAY TO SECOND *	char[11]	OCCI_SQLT_INTERVAL_DS
INTERVAL YEAR TO MONTH *	char[5]	OCCI_SQLT_INTERVAL_YM
INTERVALDS **	IntervalDS	OCCIINTERVALDS
INTERVALYM **	IntervalYM	OCCIINTERVALYM
LONG *	char[n]	OCCI_SQLT_LNG
LONG RAW *	unsigned char[n]	OCCI_SQLT_LBI
LONG VARCHAR *	char[n+siezeof(integer)]	OCCI_SQLT_LVC
LONG VARRAW *	unsigned char[n+siezeof(integer)]	OCCI_SQLT_LVB
METADATA **	MetaData	OCCIMETADATA
NAMED DATA TYPE *	struct	OCCI_SQLT_NTY
NATIVE DOUBLE *	double	OCCI_SQLT_BDOUBLE
NATIVE DOUBLE **	double	OCCIBDOUBLE
NATIVE FLOAT *	float	OCCI_SQLT_BFLOAT
NATIVE FLOAT **	float	OCCIBFLOAT
null terminated STRING *	char[n+1]	OCCI_SQLT_STR
NUMBER *	unsigned char[21]	OCCI_SQLT_NUM
NUMBER **	Number	OCCINUMBER
POBJECT **	User defined types generated by OTT utility.	OCCIPOBJECT
RAW *	unsigned char[n]	OCCI_SQLT_BIN

Table 4–2 (Cont.) External Datatypes and Corresponding C++ and OCCI Types

External Datatype	C++ Type	OCCI Type
REF *	LNOCIRef	OCCI_SQLT_REF
REF **	Ref	OCCIREF
REFANY **	RefAny	OCCIREFANY
ROWID *	LNOCIRowid	OCCI_SQLT_RID
ROWID **	Bytes	OCCIROWID
ROWID descriptor *	LNOCIRowid	OCCI_SQLT_RDD
STRING **	STL string	OCCISTRING
TIMESTAMP *	char[11]	OCCI_SQLT_TIMESTAMP
TIMESTAMP **	Timestamp	OCCITIMESTAMP
TIMESTAMP WITH LOCAL TIME ZONE *	char[7]	OCCI_SQLT_TIMESTAMP_LTZ
TIMESTAMP WITH TIME ZONE *	char[13]	OCCI_SQLT_TIMESTAMP_TZ
UNSIGNED INT *	unsigned int	OCCIUNSIGNED_INT
UNSIGNED INT **	unsigned int	OCCIUNSIGNED_INT
VARCHAR *	char[n+sizeof(short integer)]	OCCI_SQLT_VCS
VARCHAR2 *	char[n]	OCCI_SQLT_CHR
VARNUM *	char[22]	OCCI_SQLT_VNU
VARRAW *	unsigned char[n+sizeof(short integer)]	OCCI_SQLT_VBI
VECTOR **	STL vector	OCCIVECTOR

Note: The TIMESTAMP and TIMESTAMP WITH TIME ZONE datatypes are collectively known as datetimes. The INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND are collectively known as intervals.

Please note the usage of the types in the following methods of the Statement class:

- 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.
- setDataBuffer() and setDataBufferArray(): Only types of the form OCCI_SQLT_*xxx* () (for example, OCCI_SQLT_INT) in the occiCommon.h file are permitted.

Please note the usage of the types in the following methods of the ResultSet class:

- setDataBuffer() and setDataBufferArray(): Only types of the form OCCI_SQLT_*xxx* () (for example, OCCI_SQLT_INT) in the occiCommon.h file are permitted.

Description of External Datatypes

This section provides a description for each of the external datatypes.

BFILE

The external datatype 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.

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 datatype:

Example 4–1 Definition of the BDOUBLE Datatype

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

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

BFLOAT

The BFloat interface in OCCI encapsulates the native float data and the NULL information of a column or object attribute of the type binary_float. The OCCI methods in AnyData Class, ResultSet Class and Statement Class, and the global methods that take these class objects as parameters, use the following definition for the BFLOAT datatype:

Example 4–2 Definition of the BFLOAT Datatype

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

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

BLOB

The external datatype BLOB stores unstructured binary large objects. A BLOB can be thought of as a bitstream with no character set semantics. BLOBs can store up to 4 gigabytes of binary data.

BLOB datatypes have full transactional support. Changes made through OCCI participate fully in the transaction. BLOB value manipulations can be committed or rolled back. You cannot save a BLOB locator in a variable in one transaction and then use it in another transaction or session.

CHAR

The external datatype CHAR is a string of characters, with a maximum length of 2000 characters. Character strings are compared by using blank-padded comparison semantics.

CHARZ

The external datatype `CHARZ` is similar to the `CHAR` datatype, except that the string must be null terminated on input, and Oracle places a null terminator character at the end of the string on output. The null terminator serves only to delimit the string on input or output. It is not part of the data in the table.

CLOB

The external datatype `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.

DATE

The external datatype `DATE` can update, insert, or retrieve a date value using the Oracle internal seven byte date binary format, as listed in Table 4–3:

Table 4–3 Format of the DATE Datatype

	Byte 1	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**	**Byte 7**
Example	**Century**	**Year**	**Month**	**Day**	**Hour**	**Minute**	**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

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 merely signifies that it is of the CE.
- For byte 1, the second and third digits of the century are calculated as the year (an integer) divided by 100. With integer division, the fractional portion is discarded. The following calculation is for the year 1992: `1992 / 100 = 19`.
- For byte 1, `119` represents the twentieth century, 1900 to 1999. A value of `120` would represent the twenty-first century, 2000 to 2099.
- For byte 2, the second and third digits of the year are calculated as the year modulo 100: `1992 % 100 = 92`.

- For byte 2, `192` represents the ninety-second year of the current century. A value of `100` would represent the zeroth year of the current century.
- The year 2000 would yield `120` for byte 1 and `100` for byte 2.
- For bytes 3 through 7, valid dates begin at 01-JAN of the year. The month byte ranges from `1` to `12`, the date byte ranges from `1` to `31`, the hour byte ranges from `1` to `24`, the minute byte ranges from `1` to `60`, and the second byte ranges from `1` to `60`.

Example 2, 01-JAN-4712 BCE:

- For years prior to 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`.

Notes:

- 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 datatype `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 datatype `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` datatype.
- 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 datatype is different from `OCCI DATE` which corresponds to a C++ `Date` datatype.

FLOAT

The external datatype FLOAT processes numbers with fractional parts. The number is represented in the host system's floating-point format. Normally, the length is 4 or 8 bytes.

The internal format of an Oracle number is decimal. Most floating-point implementations are binary. Oracle, therefore, represents numbers with greater precision than floating-point representations.

INTEGER

The external datatype 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.

Note: A rounding error may occur when converting between FLOAT and NUMBER. Using a FLOAT as a bind variable in a query may return an error. You can work around this by converting the FLOAT to a string and using the OCCI type OCCI_SQLT_CHR or the OCCI type OCCI_SQLT_STR for the operation.

INTERVAL DAY TO SECOND

The external datatype INTERVAL DAY TO SECOND stores the difference between two datetime values in terms of days, hours, minutes, and seconds. Specify this datatype 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 precisions, you must specify the precisions in the literal. For example, you might specify an interval of 100 days, 10 hours, 20 minutes, 42 seconds, and 22 hundredths of a second as follows:

```
INTERVAL '100 10:20:42.22' DAY(3) TO SECOND(2)
```

You can also use abbreviated forms of the INTERVAL DAY TO SECOND literal. For example:

- INTERVAL '90' MINUTE maps to INTERVAL '00 00:90:00.00' DAY TO SECOND(2)
- INTERVAL '30:30' HOUR TO MINUTE maps to INTERVAL '00 30:30:00.00' DAY TO SECOND(2)
- INTERVAL '30' SECOND(2,2) maps to INTERVAL '00 00:00:30.00' DAY TO SECOND(2)

INTERVAL YEAR TO MONTH

The external datatype INTERVAL YEAR TO MONTH stores the difference between two datetime values by using the YEAR and MONTH datetime fields. Specify INTERVAL YEAR TO MONTH as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

The placeholder *year_precision* is the number of digits in the YEAR datetime field. The default value of *year_precision* is 2. To specify an INTERVAL YEAR TO MONTH literal with a nondefault *year_precision*, you must specify the precision in the literal. For example, the following INTERVAL YEAR TO MONTH literal indicates an interval of 123 years, 2 months:

```
INTERVAL '123-2' YEAR(3) TO MONTH
```

You can also use abbreviated forms of the INTERVAL YEAR TO MONTH literal. For example,

- INTERVAL '10' MONTH maps to INTERVAL '0-10' YEAR TO MONTH
- INTERVAL '123' YEAR(3) maps to INTERVAL '123-0' YEAR(3) TO MONTH

LONG

The external datatype LONG stores character strings longer than 4000 bytes and up to 2 gigabytes in a column of datatype LONG. Columns of this type are only used for storage and retrieval of long strings. They cannot be used in methods, expressions, or `WHERE` clauses. `LONG` column values are generally converted to and from character strings.

LONG RAW

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

LONG VARCHAR

The external datatype `LONG VARCHAR` stores data from and into an Oracle `LONG` column. The first four bytes contain the length of the item. The maximum length of a `LONG VARCHAR` is `2` gigabytes.

LONG VARRAW

The external datatype `LONG VARRAW` store data from and into an Oracle `LONG RAW` column. The length is contained in the first four bytes. The maximum length is `2` gigabytes.

NCLOB

The external datatype `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.

`NCLOB`s have full transactional support. Changes made through OCCI participate fully in the transaction. `NCLOB` value manipulations can be committed or rolled back. You cannot save an `NCLOB` locator in a variable in one transaction and then use it in another transaction or session.

You cannot create an object with `NCLOB` attributes, but you can specify `NCLOB` parameters in methods.

NUMBER

You should not need to use `NUMBER` as an external datatype. If you do use it, Oracle returns numeric values in its internal `21`-byte binary format and will expect this format on input. The following discussion is included for completeness only.

Oracle stores values of the `NUMBER` datatype in a variable-length format. The first byte is the exponent and is followed by `1` to `20` mantissa bytes. The high-order bit of the exponent byte is the sign bit; it is set for positive numbers and it is cleared for negative numbers. The lower `7` bits represent the exponent, which is a base-`100` digit with an offset of `65`.

To calculate the decimal exponent, add `65` to the base-`100` exponent and add another `128` if the number is positive. If the number is negative, you do the same, but subsequently the bits are inverted. For example, `-5` has a base-`100` exponent = `62 (0x3e)`. The decimal exponent is thus `(~0x3e)-128-65 = 0xc1-128-65 = 193-128-65 = 0`.

Each mantissa byte is a base-`100` digit, in the range `1` to `100`. For positive numbers, the digit has `1` added to it. So, the mantissa digit for the value `5` is `6`. For negative numbers, instead of adding 1, the digit is subtracted from `101`. So, the mantissa digit for the number `-5` is: `101-5 = 96`. Negative numbers have a byte containing `102` appended to the data bytes. However, negative numbers that have `20` mantissa bytes do not have the trailing `102` byte. Because the mantissa digits are stored in base-`100`, each byte can represent two decimal digits. The mantissa is normalized; leading zeroes are not stored.

Up to `20` data bytes can represent the mantissa. However, only `19` are guaranteed to be accurate. The `19` data bytes, each representing a base-`100` digit, yield a maximum precision of `38` digits for an internal datatype `NUMBER`.

Note that this datatype is different from `OCCI NUMBER` which corresponds to a C++ `Number` datatype.

OCCI BFILE

See Also: Chapter 10, "OCCI Application Programming Interface", Bfile Class on page 10-26

OCCI BLOB

See Also: Chapter 10, "OCCI Application Programming Interface", Blob Class on page 10-35

OCCI BYTES

See Also: Chapter 10, "OCCI Application Programming Interface", Bytes Class on page 10-45

OCCI CLOB

See Also: Chapter 10, "OCCI Application Programming Interface", Clob Class on page 10-48

OCCI DATE

See Also: Chapter 10, "OCCI Application Programming Interface", Date Class on page 10-89

OCCI INTERVALDS

See Also: Chapter 10, "OCCI Application Programming Interface", IntervalDS Class on page 10-118

OCCI INTERVALYM

See Also: Chapter 10, "OCCI Application Programming Interface", IntervalYM Class on page 10-131

OCCI NUMBER

See Also: Chapter 10, "OCCI Application Programming Interface", Number Class on page 10-168

OCCI POBJECT

See Also: Chapter 10, "OCCI Application Programming Interface", PObject Class on page 10-194

OCCI REF

See Also: Chapter 10, "OCCI Application Programming Interface", Ref Class on page 10-207

OCCI REFANY

See Also: Chapter 10, "OCCI Application Programming Interface", RefAny Class on page 10-214

OCCI STRING

The external datatype OCCI STRING corresponds to an STL string.

OCCI TIMESTAMP

> **See Also:** Chapter 10, "OCCI Application Programming Interface", Timestamp Class on page 10-327

OCCI VECTOR

The external datatype OCCI VECTOR is used to represent collections, for example, a nested table or VARRAY. CREATE TYPE num_type as VARRAY OF NUMBER(10) can be represented in a C++ application as vector<int>, vector<Number>, and so on.

RAW

The external datatype RAW is used for binary data or byte strings that are not to be interpreted or processed by Oracle. RAW could be used, for example, for graphics character sequences. The maximum length of a RAW column is 2000 bytes.

When RAW data in an Oracle table is converted to a character string, the data is represented in hexadecimal code. Each byte of RAW data is represented as two characters that indicate the value of the byte, ranging from 00 to FF. If you input a character string by using RAW, then you must use hexadecimal coding.

REF

The external datatype REF is a reference to a named datatype. To allocate a REF for use in an application, declare a variable as a pointer to a REF.

ROWID

The external datatype ROWID identifies a particular row in a database table. The ROWID is often returned from a query by issuing a statement similar to the following example:

```
SELECT ROWID, var1, var2 FROM db;
```

You can then use the returned ROWID in further DELETE statements.

If you are performing a SELECT for an UPDATE operation, then the ROWID is implicitly returned.

STRING

The external datatype `STRING` behaves like the external datatype `VARCHAR2` (datatype code 1), except that the external datatype `STRING` must be null-terminated.

Note that this datatype is different from `OCCI STRING` which corresponds to a C++ STL string datatype.

TIMESTAMP

The external datatype `TIMESTAMP` is an extension of the `DATE` datatype. It stores the year, month, and day of the `DATE` datatype, plus hour, minute, and second values. Specify the `TIMESTAMP` datatype 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 datatype is different from `OCCI TIMESTAMP`.

TIMESTAMP WITH LOCAL TIME ZONE

The external datatype `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` datatype 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.

TIMESTAMP WITH TIME ZONE

The external datatype `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` datatype 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'
```

UNSIGNED INT

The external datatype `UNSIGNED INT` is used for unsigned binary integers. The size in bytes is operating system dependent. The host system architecture determines the order of the bytes in a word. If the number being output from Oracle is not an integer, the fractional part is discarded, and no error is returned. If the number to be returned exceeds the capacity of an unsigned integer for the operating system, Oracle returns an overflow on conversion error.

VARCHAR

The external datatype `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 datatype.

VARCHAR2

The external datatype `VARCHAR2` is a variable-length string of characters up to `4000` bytes.

VARNUM

The external datatype `VARNUM` is similar to the external datatype `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 4–4 VARNUM Examples

Decimal Value	Length Byte	Exponent Byte	Mantissa Bytes	Terminator Byte
0	1	128	N/A	N/A
5	2	193	6	N/A
-5	3	62	96	102
2767	3	194	28, 68	N/A
-2767	4	61	74, 34	102
100000	2	195	11	N/A
1234567	5	196	2, 24, 46, 68	N/A

VARRAW

The **external** datatype VARRAW is similar to the external datatype 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 datatype.

NATIVE DOUBLE

This **external** datatype implements the IEEE 754 standard double-precision floating point datatype. It is represented in the host system's native floating point format. The datatype 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 datatypes used by RDBMS Clients
- Supports a wider range of values used in scientific calculations.

NATIVE FLOAT

This **external** datatype implements the IEEE 754 single-precision floating point datatype. It is represented in the host system's native floating point format. The datatype 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 datatypes used by RDBMS Clients
- Supports a wider range of values used in scientific calculations

Data Conversions

Table 4–5 lists the supported conversions from Oracle internal datatypes to external datatypes, and from external datatypes to internal column representations. Note the following conditions:

- A REF stored in the database is converted to OCCI_SQLT_REF on output
- OCCI_SQLT_REF is converted to the internal representation of a REF on input
- A named datatype stored in the database is converted to OCCI_SQLT_NTY (and represented by a C structure in the application) on output
- OCCI_SQLT_NTY (represented by a C structure in an application) is converted to the internal representation of the corresponding datatype on input
- A LOB and a BFILE are represented by descriptors in OCCI applications, so there are no input or output conversions

Table 4–5 Data Conversions Between External and Internal datatypes

	Internal Datatypes									
External Datatypes	**VARCHAR 2**	**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	-	-
DATE	I/O	-	I	-	I/O	-	-	I/O	-	-
DECIMAL	I/O[4]	I/O	I	-	-	-	-	I/O[4]	-	-
FLOAT	I/O[4]	I/O	I	-	-	-	-	I/O[4]	I/O	I/O
INTEGER	I/O[4]	I/O	I	-	-	-	-	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]	-	I[5, 6]	-	-	I/O	I/O	O[6]	-	-
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

Table 4–5 (Cont.) Data Conversions Between External and Internal datatypes

	Internal Datatypes									
External Datatypes	**VARCHAR 2**	**NUMBER**	**LONG**	**ROWID**	**DATE**	**RAW**	**LONG RAW**	**CHAR**	**BFLOAT**	**BDOUBLE**
`LONG VARRAW`	I/O[6]	-	I[5, 6]	-	-	I/O	I/O	I/O[6]	-	-
`NUMBER`	I/O[4]	I/O	I	-	-	-	-	I/O[4]	I/O	I/O
`OCCI BDouble`	I/O	1/O	I	-	-	-	-	I/O	I/O	I/O
`OCCI BFloat`	I/O	1/O	I	-	-	-	-	I/O	I/O	I/O
`OCCI Bytes`	I/O[6]	-	I[5, 6]	-	-	I/O	I/O	I/O[6]	-	-
`OCCI Date`	I/O	-	I	-	I/O	-	-	I/O	-	-
`OCCI Number`	I/O[4]	I/O	I	-	-	-	-	I/O[4]	I/O	I/O
`OCCI Timestamp`	-	-	-	-	-	-	-	-	-	-
`RAW`	I/O[6]	-	I[5, 6]	-	-	I/O	I/O	I/O[6]	-	-
`ROWID`	I	-	I	I/O	-	-	-	I	-	-
`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	-	-	-	-	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]	-	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	-	-	-	-	I/O[4]	I/O	I/O
`VARRAW`	I/O[6]	-	I[5, 6]	-	-	I/O	I/O	I/O[6]	-	-

Note: Conversions valid for I (Input only), O (Output Only), I/O (Input or Output)

1. Must be in Oracle ROWID format for input; returned in Oracle ROWID format on output.
2. Must be in Oracle DATE format for input; returned in Oracle DATE format on output.
3. Must be in hexadecimal format for input; returned in hexadecimal format on output.
4. Must represent a valid number for output.
5. Length must be less than or equal to 2000 characters.
6. Stored in hexadecimal format on output; must be in hexadecimal format on output.

Data Conversions for LOB Datatypes

Table 4–6 Data Conversions for LOBs

	INTERNAL DATATYPES	
EXTERNAL DATATYPES	**CLOB**	**BLOB**
`VARCHAR`	I/O	-
`CHAR`	I/O	-
`LONG`	I/O	-
`LONG VARCHAR`	I/O	-
`STL String`	I/O	-
`RAW`	-	I/O
`VARRAW`	-	I/O
`LONG RAW`	-	I/O
`LONG VARRAW`	-	I/O
`OCCI Bytes`	-	I/O

Data Conversions for Date, Timestamp, and Interval Datatypes

You can also use one of the character data types for the host variable used in a fetch or insert operation from or to a datetime or interval column. Oracle will do the conversion between the character data type and datetime/interval data type for you.

Table 4–7 Data Conversions for Date, Timestamp, and Interval Datatypes

External Types	Internal Types						
	VARCHAR, CHAR	DATE	TS	TSTZ	TSLTZ	INTERVAL YEAR TO MONTH	INTERVAL DAY TO SECOND
`VARCHAR2, CHAR`	I/O	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	-	-
`OCCI Date`	I/O	I/O	I/O	I/O	I/O	-	-
`ANSI DATE`	I/O	I/O	I/O	I/O	I/O	-	-
`TIMESTAMP (TS)`	I/O	I/O	I/O	I/O	I/O	-	-
`OCCI Timestamp`	I/O	I/O	I/O	I/O	I/O	-	-
`TIMESTAMP WITH TIME ZONE (TSTZ)`	I/O	I/O	I/O	I/O	I/O	-	-
`TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)`	I/O	I/O	I/O	I/O	I/O	-	-
`INTERVAL YEAR TO MONTH`	I/O	-	-	-	-	I/O	-
`OCCI IntervalYM`	I/O	-	-	-	-	I/O	-
`INTERVAL DAY TO SECOND`	I/O	-	-	-	-	-	I/O
`OCCI IntervalDS`	I/O	-	-	-	-	-	I/O

These consideration apply when converting between Date, Timestamp and Interval datatypes:

- When assigning a source with time zone to a target without a time zone, the time zone portion of the source is ignored. On assigning a source without a time zone to a target with a time zone, the time zone of the target is set to the session's default time zone.
- When assigning an Oracle `DATE` to a `TIMESTAMP`, the `TIME` portion of the `DATE` is copied over to the `TIMESTAMP`. When assigning a `TIMESTAMP` to Oracle

DATE, the TIME portion of the result DATE is set to zero. This is done to encourage migration of Oracle DATE to ANSI compliant DATETIME data types.

- (When assigning an ANSI DATE to an Oracle DATE or a TIMESTAMP, the TIME portion of the Oracle DATE and the TIMESTAMP are set to zero. When assigning an Oracle DATE or a TIMESTAMP to an ANSI DATE, the TIME portion is ignored.
- When assigning a DATETIME to a character string, the DATETIME is converted using the session's default DATETIME format. When assigning a character string to a DATETIME, the string must contain a valid DATETIME value based on the session's default DATETIME format.
- When assigning a character string to an INTERVAL, the character string must be a valid INTERVAL character format.
- When converting from TSLTZ to CHAR, DATE, TIMESTAMP and TSTZ, the value will be adjusted to the session time zone.
- When converting from CHAR, DATE, and TIMESTAMP to TSLTZ, the session time zone will be stored in memory.
- When assigning TSLTZ to ANSI DATE, the time portion will be 0.
- When converting from TSTZ, the time zone which the time stamp is in will be stored in memory.
- When assigning a character string to an interval, the character string must be a valid interval character format.

5

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

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:

- Tables
- Types
- Sequences
- Views
- Type Attributes
- Columns
- Procedures

- Type Methods
- Arguments
- Functions
- Collections
- Results
- Packages
- Synonyms
- 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 already 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 described information. The `MetaData` class provides the `get`*`xxx`*`()` methods to access this information.

Notes on Types and Attributes

When performing `DESCRIBE` operations, be aware of the following issues:

- The `ATTR_TYPECODE` returns typecodes that represent the type supplied when you created a new type by using the `CREATE TYPE` statement. These typecodes are of the enumerated type `TypeCode`, which are represented by `OCCI_TYPECODE` constants.

  Note: Internal PL/SQL types (boolean, indexed table) are not supported.

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

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. To do this, you must 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 sutures containing more information. More information about an object can be obtained by calling the `get`*`xxx`*`()` methods.

If you need to 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. Since you are aware of the type of the object that you want to describe (table, column, type, collection, function, procedure, and so on), you can obtain the attribute list as shown in Table 5–1. You can retrieve the value into a variable of the type specified in the table by calling the corresponding `get`*`xxx`*`()` method.

Table 5–1 Attribute Groupings

Attribute Type	Description
Parameter Attributes on page 5-8	Attributes belonging to all elements
Table and View Attributes on page 5-9	Attributes belonging to tables and views
Procedure, Function, and Subprogram Attributes on page 5-10	Attributes belonging to procedures, functions, and package subprograms
Package Attributes on page 5-11	Attributes belonging to packages
Type Attributes on page 5-11	Attributes belonging to types
Type Attribute Attributes on page 5-13	Attributes belonging to type attributes
Type Method Attributes on page 5-14	Attributes belonging to type methods
Collection Attributes on page 5-15	Attributes belonging to collection types

Table 5–1 (Cont.) Attribute Groupings

Attribute Type	**Description**
Synonym Attributes on page 5-16	Attributes belonging to synonyms
Sequence Attributes on page 5-16	Attributes belonging to sequences
Column Attributes on page 5-17	Attributes belonging to columns of tables or views
Argument and Result Attributes on page 5-18	Attributes belonging to arguments / results
List Attributes on page 5-19	Attributes that designate the list type
Schema Attributes on page 5-20	Attributes specific to schemas
Database Attributes on page 5-20	Attributes specific to databases

Metadata Code Examples

This section provides code examples for obtaining:

- `Connection` metadata
- `ResultSet` metadata

Example 5–1 How to Obtain Metadata About Attributes of a Simple Database Table

The following code 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)
   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,for example 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 5–2 How to Obtain Metadata from a Column Containing User Defined Types

```
/* 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 the CUSTOMERS table; call the getxxx methods */
/* Call getString */
cout<<"Schema:"<<(custtab_metaData.getString(MetaData::ATTR_OBJ_SCHEMA))<<endl;
if(custtab_metaData.getInt(custtab_metaData::ATTR_PTYPE)==MetaData::PTYPE_TABLE)
   cout<<"CUSTOMERS is a table"<<endl;
else
   cout<<"CUSTOMERS is not a table"<<endl;

/* Call getVector to obtain list of columns in the CUSTOMERS table */
```

```
vector<MetaData>listOfColumns;
listOfColumns=custtab_metaData.getVector(MetaData::ATTR_LIST_COLUMNS);

/* Assuming metadata for column cust_address_typ is fourth element in list*/
MetaData customer_address=listOfColumns[3];

/* Obtain the metadata for the customer_address attribute */
int typcode = customer_address.getInt(MetaData::ATTR_TYPECODE);
if(typcode==OCCI_TYPECODE_OBJECT)
   cout<<"customer_address is an object type"<<endl;
else
   cout<<"customer_address is not an object type"<<endl;

string objectName=customer_address.getString(MetaData::ATTR_OBJ_NAME);

/* Now that you have the name of the address object,
   the metadata of the attributes of the type can be obtained by using
   getMetaData on the connection by passing the object name
*/
MetaData address = connection->getMetaData(objectName);

/* Call getVector to obtain the list of the address object attributes */
vector<MetaData> attributeList =
      address.getVector(MetaData::ATT_LIST_TYPE_ATTRS);

/* and so on to obtain metadata on other address object specific attributes */
```

Example 5–3 How to obtain object metadata from a reference

The following code example demonstrates how to obtain metadata about an object when using a reference to it:

Assuming the following schema structure:

```
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 datatype 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 datatype is a reference, get the Ref and obtain the metadata
      about the object by passing the Ref to getMetaData */
   if(dataType==SQLT_REF)
      RefAny refTdo=colList[i].getRef(MetaData::ATTR_REF_TDO);

   /* Now you can obtain the metadata about the object as shown
   MetaData tdo_metaData=connection->getMetaData(refTdo);

   /* Now that you have the metadata about the TDO, you can obtain the metadata
      about the object */
}
```

Example 5–4 How to Obtain Metadata About a Select List from a ResultSet Object

```
/* 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.

Attribute Reference

This section describes the attributes belonging to schema and subschema objects.

Parameter Attributes

All elements have some attributes specific to that element and some generic attributes. Table 5–2 describes the attributes that belong to all elements:

Table 5–2 Attributes that Belong to All Elements

Attribute	Description	Attribute Datatype
`ATTR_OBJ_ID`	Object or schema ID	`unsigned int`
`ATTR_OBJ_NAME`	Object, schema, or database name	`string`

Table 5–2 (Cont.) Attributes that Belong to All Elements

Attribute	Description	Attribute Datatype
ATTR_OBJ_SCHEMA	Schema where object is located	string
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.

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 5–3:

Table 5–3 Attributes that Belong to Tables or Views

Attribute	Description	Attribute Datatype
ATTR_OBJID	Object ID	unsigned int
ATTR_NUM_COLS	Number of columns	int
ATTR_LIST_COLUMNS	Column list (type PTYPE_LIST)	vector<MetaData>

Table 5–3 (Cont.) Attributes that Belong to Tables or Views

Attribute	Description	Attribute Datatype
`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: ■ `OCCI_DURATION_SESSION (session)` ■ `OCCI_DURATION_TRANS (transaction)` ■ `OCCI_DURATION_NULL (table not temporary)`	`int`

The additional attributes belonging to tables are described in Table 5–4.

Table 5–4 Attributes Specific to Tables

Attribute	Description	Attribute Datatype
`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`

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 5–5.

Table 5–5 Attributes that Belong to Procedures or Functions

Attribute	Description	Attribute Datatype
`ATTR_LIST_ARGUMENTS`	Argument list; refer to List Attributes on page 5-19.	`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 5–6.

Table 5–6 Attributes that Belong to Package Subprograms

Attribute	Description	Attribute Datatype
`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`

Package Attributes

A parameter for a package (type `PTYPE_PKG`) has the type-specific attributes described in Table 5–7.

Table 5–7 Attributes that Belong to Packages

Attribute	Description	Attribute Datatype
`ATTR_LIST_SUBPROGRAMS`	Subprogram list; refer to List Attributes on page 5-19.	`vector<MetaData>`
`ATTR_IS_INVOKER_RIGHTS`	Identifies whether the package has invoker's rights	`bool`

Type Attributes

A parameter for a type (type `PTYPE_TYPE`) has attributes described in Table 5–8.

Table 5–8 Attributes that Belong to Types

Attribute	Description	Attribute Datatype
`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`	Typecode. Can be: ■ `OCCI_TYPECODE_OBJECT` ■ `OCCI_TYPECODE_NAMEDCOLLECTION` Refer to Notes on Types and Attributes on page 5-2.	`int`

Table 5–8 (Cont.) Attributes that Belong to Types

Attribute	Description	Attribute Datatype
ATTR_COLLECTION_TYPECODE	Typecode of collection if type is collection; invalid otherwise. Can be: ▪ OCCI_TYPECODE_VARRAY ▪ OCCI_TYPECODE_TABLE Refer to Notes on Types and Attributes on page 5-2.	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
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 on page 5-15	MetaData
ATTR_NUM_TYPE_ATTRS	Number of type attributes	unsigned int
ATTR_LIST_TYPE_ATTRS	List of type attributes Refer to List Attributes on page 5-19	vector<MetaData>
ATTR_NUM_TYPE_METHODS	Number of type methods	unsigned int

Table 5–8 (Cont.) Attributes that Belong to Types

Attribute	Description	Attribute Datatype
`ATTR_LIST_TYPE_METHODS`	List of type methods Refer to List Attributes on page 5-19	`vector<MetaData>`
`ATTR_MAP_METHOD`	Map method of type Refer to Type Method Attributes on page 5-14	`MetaData`
`ATTR_ORDER_METHOD`	Order method of type; refer to Type Method Attributes on page 5-14	`MetaData`

Type Attribute Attributes

A parameter for an attribute of a type (type `PTYPE_TYPE_ATTR`) has the attributes described in Table 5–9.

Table 5–9 Attributes that Belong to Type Attributes

Attribute	Description	Attribute Datatype
`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 Refer to Notes on Types and Attributes on page 5-2.	`int`
`ATTR_DATA_TYPE`	Datatype of the type attribute Refer to Notes on Types and Attributes on page 5-2.	`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`
`ATTR_TYPE_NAME`	A string that is the type name. The returned value will contain the type name if the datatype is `SQLT_NTY` or `SQLT_REF`. If the datatype is `SQLT_NTY`, then the name of the named datatype's type is returned. If the datatype is `SQLT_REF`, then the type name of the named datatype pointed to by the `REF` is returned.	`string`

Table 5–9 (Cont.) Attributes that Belong to Type Attributes

Attribute	Description	Attribute Datatype
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

Type Method Attributes

A parameter for a method of a type (type PTYPE_TYPE_METHOD) has the attributes described in Table 5–10.

Table 5–10 Attributes that Belong to Type Methods

Attribute	Description	Attribute Datatype
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

Table 5–10 (Cont.) Attributes that Belong to Type Methods

Attribute	Description	Attribute Datatype
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

Collection Attributes

A parameter for a collection type (type PTYPE_COLL) has the attributes described in Table 5–11.

Table 5–11 Attributes that Belong to Collection Types

Attribute	Description	Attribute Datatype
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; refer to Notes on Types and Attributes on page 5-2.	int
ATTR_DATA_TYPE	The datatype of the type attribute; refer to Notes on Types and Attributes on page 5-2.	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

Table 5–11 (Cont.) Attributes that Belong to Collection Types

Attribute	Description	Attribute Datatype
`ATTR_TYPE_NAME`	String that is the type name. The returned value will contain the type name if the datatype is `SQLT_NTY` or `SQLT_REF`. If the datatype is `SQLT_NTY`, then the name of the named datatype's type is returned. If the datatype is `SQLT_REF`, then the type name of the named datatype 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.	`RefAny`
`ATTR_CHARSET_ID`	Typecode; refer to Notes on Types and Attributes on page 5-2.	`int`
`ATTR_CHARSET_FORM`	The datatype of the type attribute; refer to Notes on Types and Attributes on page 5-2.	`int`

Synonym Attributes

A parameter for a synonym (type `PTYPE_SYN`) has the attributes described in Table 5–12.

Table 5–12 Attributes that Belong to Synonyms

Attribute	Description	Attribute Datatype
`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`

Sequence Attributes

A parameter for a sequence (type `PTYPE_SEQ`) has the attributes described in Table 5–13.

Table 5–13 Attributes that Belong to Sequences

Attribute	Description	Attribute Datatype
`ATTR_OBJID`	Object ID	`unsigned int`
`ATTR_MIN`	Minimum value (in Oracle number format)	`Number`

Table 5–13 (Cont.) Attributes that Belong to Sequences

Attribute	Description	Attribute Datatype
`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`

Column Attributes

A parameter for a column of a table or view (type `PTYPE_COL`) has the attributes described in Table 5–14.

Table 5–14 Attributes that Belong to Columns of Tables or Views

Attribute	Description	Attribute Datatype
`ATTR_DATA_SIZE`	Column length in codepoints. The number of codepoints allowed in the column.	`int`
`ATTR_DATA_TYPE`	Type of length semantics of the column. Valid values are `0` for byte-length semantics and `1` for codepoint-length semantics.	`int`
`ATTR_NAME`	Maximum size of the column. This length is returned in bytes and not characters for strings and raws. Returns `22` for `NUMBER`.	`string`
`ATTR_PRECISION`	The datatype of the column; refer to Notes on Types and Attributes on page 5-2.	`int`
`ATTR_SCALE`	Pointer to a string that is the column name	`int`
`ATTR_IS_NULL`	Returns `TRUE` if the attribute is `NULL`; `FALSE` otherwise.	`bool`
`ATTR_TYPE_NAME`	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`.	`string`
`ATTR_SCHEMA_NAME`	Returns `0` if null values are not permitted for the column	`string`
`ATTR_REF_TDO`	Returns a string that is the type name. The returned value will contain the type name if the datatype is `SQLT_NTY` or `SQLT_REF`. If the datatype is `SQLT_NTY`, then the name of the named datatype's type is returned. If the datatype is `SQLT_REF`, then the type name of the named datatype pointed to by the `REF` is returned.	`RefAny`

Table 5–14 (Cont.) Attributes that Belong to Columns of Tables or Views

Attribute	Description	Attribute Datatype
`ATTR_CHARSET_ID`	Returns a string with the schema name under which the type has been created.	`int`
`ATTR_CHARSET_FORM`	The `REF` of the TDO for the type, if the column type is an object type	`int`

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 5–15.

Table 5–15 Attributes that Belong to Arguments / Results

Attribute	Description	Attribute Datatype
`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; refer to Notes on Types and Attributes on page 5-2.	`int`
`ATTR_DATA_TYPE`	Datatype of the argument; refer to Notes on Types and Attributes on page 5-2.	`int`
`ATTR_DATA_SIZE`	Size of the datatype 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`	Datatype 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>`

Table 5–15 (Cont.) Attributes that Belong to Arguments / Results

Attribute	Description	Attribute Datatype
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 in the case of package local types. The returned value contains the type name if the datatype is SQLT_NTY or SQLT_REF. If the datatype is SQLT_NTY, then the name of the named datatype's type is returned. If the datatype is SQLT_REF, then the type name of the named datatype 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 under which the package was created in the case of package local types	string
ATTR_SUB_NAME	For SQLT_NTY or SQLT_REF, returns a string with the type name, in the case of package local types	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 in the case of package local 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

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 5–16 displays the list attributes.

Table 5–16 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

Schema Attributes

A parameter for a schema type (type PTYPE_SCHEMA) has the attributes described in Table 5–17.

Table 5–17 Attributes Specific to Schemas

Attribute	Description	Attribute Datatype
ATTR_LIST_OBJECTS	List of objects in the schema	string

Database Attributes

A parameter for a database (type PTYPE_DATABASE) has the attributes described in Table 5–18.

Table 5–18 Attributes Specific to Databases

Attribute	Description	Attribute Datatype
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

Table 5–18 (Cont.) Attributes Specific to Databases

Attribute	Description	Attribute Datatype
`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 still reexecute the statement 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`

6 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:

> **See Also:** `$ORACLE_HOME/rdbms/demo` for a complete code listing of the demonstration program used in this chapter and the class and method implementation generated by the OTT utility.

Overview of the Object Type Translator Utility

The Object Type Translator (OTT) utility assists in the development of applications that make use of user-defined types in an Oracle database server.

Through the use of SQL `CREATE TYPE` statements, you can create object types. The definitions of these types are stored in the database and can be used in the creation of 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
- 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

Using the OTT Utility

To translate database types to C++ representation, you must explicitly invoke the OTT utility. OCCI programmers must register the mappings with the environment. This function is generated by the OTT utility.

On most operating systems, the OTT utility is invoked on the command line. It takes as input an `INTYPE` file, and generates an `OUTTYPE` file, one or more C++ header files that contain the prototype information, and additional C++ method files that register generated mappings.

Example 6–1 How to Use the OTT Utility

The following command invokes the OTT utility and generates C++ classes:

```
ott userid=scott/tiger intype=demoin.typ outtype=demoout.typ code=cpp
   hfile=demo.h cppfile=demo.cpp mapfile=RegisterMappings.cpp
```

This command causes the OTT utility to connect to the database as username `scott` with password `tiger`, and use the `demoin.typ` file as the `INTYPE` file, and the `demoout.typ` file as the `OUTTYPE` file. The resulting declarations are output to the file `demo.h` in C++, specified by the `CODE=cpp` parameter, the method implementations written to the file `demo.cpp`, and the functions to register mappings is written to `RegisterMappings.cpp` with its prototype written to `RegisterMappings.h`.

> **See Also:** Extending C++ Classes on page 6-36 for a complete C++ example

Creating Types in the Database

The first step in using the OTT utility is to create object types or named collection types and store them in the database. This is accomplished through the use of the SQL `CREATE TYPE` statement.

Example 6–2 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_l ADDRESS_TAB) NOT FINAL;
CREATE TYPE STUDENT UNDER PERSON (school_name CHAR(20));
```

Invoking the OTT Utility

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

Specifying OTT Parameters

You can specify OTT parameters either on the command line or in a configuration file. Certain parameters can also be specified in the `INTYPE` file.

If you specify a parameter in more than one place, then its value on the command line takes precedence over its value in the `INTYPE` file. The value in the `INTYPE` file takes precedence over its value in a user-defined configuration file, which takes precedence over its value in the default configuration file.

Parameter precedence then is as follows:

1. OTT command line
2. Value in `INTYPE` file
3. User-defined configuration file
4. Default configuration file

For global options (that is, options on the command line or options at the beginning of the `INTYPE` file before any `TYPE` statements), the value on the command line overrides the value in the `INTYPE` file. (The options that can be specified globally in the `INTYPE` file are `CASE`, `INITFILE`, `INITFUNC`, `MAPFILE` and `MAPFUNC`, but not `HFILE` or `CPPFILE`.) Anything in the `INTYPE` file in a `TYPE` specification applies to a particular type only and overrides anything on the command line that would otherwise apply to the type. So if you enter `TYPE person HFILE=p.h`, then it applies to `person` only and overrides the `HFILE` on the command line. The statement is not considered a command line parameter.

Setting Parameters on the Command Line

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

Setting Parameters in the INTYPE File

The `INTYPE` file gives a list of types for the OTT utility to translate.

The parameters `CASE`, `CPPFILE`, `HFILE`, `INITFILE`, `INITFUNC`, `MAPFILE`, and `MAPFUNC` can appear in the `INTYPE` file.

Setting Parameters in the Configuration File

A configuration file is a text file that contains OTT parameters. Each nonblank line in the file contains one parameter, with its associated value or values. If more than one parameter is put on a line, then only the first one will be used. No blank space is allowed on any nonblank line of a configuration file.

A configuration file can be named on the command line. In addition, a default configuration file is always read. This default configuration file must always exist, but can be empty. The name of the default configuration file is `ottcfg.cfg`, and the location of the file is operating system-specific.

See Also: Your operating system-specific documentation for more information about the location of the default configuration file.

Invoking the OTT Utility on the Command Line

On most platforms, the OTT utility is invoked on the command line. You can specify the input and output files and the database connection information at the command line, among other things.

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

Example 6–3 How to Invoke the OTT Utility to Generate C++ Classes

```
ott userid=scott/tiger intype=demoin.typ outtype=demoout.typ code=cpp
   hfile=demo.h cppfile=demo.cpp mapfile=RegisterMappings.cpp
```

Caution: No spaces are permitted around the equals sign (=) on the OTT command line.

An OTT command line statement consists of the command OTT, followed by a list of OTT utility parameters.

The `HFILE` parameter is almost always used. If omitted, then `HFILE` must be specified individually for each type in the `INTYPE` file. If the OTT utility determines that a type not listed in the `INTYPE` file must be translated, then an error will be 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 will be translated. See the parameter descriptions in the following section for more information.

Elements Used on the OTT Command Line

Elements used on the OTT command line are:

- OTT command that invokes the OTT utility. It must be the first item on the command line.
- USERID parameter, described in detail on page 6-15.
- INTYPE parameter, described in detail on page 6-9.
- OUTTYPE parameter, described in detail on page 6-10.
- CODE parameter, described in detail on page 6-8.
- HFILE parameter, described in detail on page 6-9.
- CPPFILE parameter, described in detail on page 6-8.
- MAPFILE parameter, described in detail on page 6-10.

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 6–1 lists all OTT Utility parameters:

Table 6–1 Summary of OTT Utility Parameters

Parameter	Description
ATTRACCESS	Specifies whether the access to type attributes will be 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.
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.

Table 6–1 (Cont.) Summary of OTT Utility Parameters

Parameter	Description
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 will use.

ATTRACCESS

This parameter specifies access to type attributes:

- `PROTECTED` -- default.
- `PRIVATE` -- the OTT utility generates accessory and mutator methods for each type attribute, `get`*`XXX`*`()` and `set`*`XXX`*`()`.

CASE

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

- `SAME` -- the case of letters remains unchanged when converting database type and attribute names to C++ identifiers.
- `LOWER` -- all uppercase letters are converted to lowercase.
- `UPPER` -- all lowercase letters are converted to uppercase.
- `OPPOSITE` -- 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.

Note: Case insensitive SQL identifiers not mentioned in the `INTYPE` file will appear in uppercase if `CASE=SAME`, and in lowercase if `CASE=OPPOSITE`. A SQL identifier is case insensitive if it was not quoted when it was declared.

CODE

This parameter specifies the host language to be output by the OTT utility. `CODE=CPP` must be specified for the OTT utility to generate C++ code for OCCI applications.

CONFIG

This parameter specifies the name of the OTT configuration file that lists commonly used parameter specifications. Parameter specifications are also read from a system configuration file found in an operating system-dependent location. All remaining parameter specifications must appear either on the command line or in the `INTYPE` file.

Note: The `CONFIG` parameter can only be specified on the OTT command line. It is not allowed in the `CONFIG` file.

CPPFILE

This parameter specifies the name of the C++ source file that will contain 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.

ERRTYPE

This parameter specifies the name of the error message output file. Information and error messages are sent to the standard output whether or not 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 will include 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 such as .TLS or .tls is added automatically.

HFILE

This parameter specifies the name of the header (.h) file to be generated by the OTT utility. The HFILE specified on the command line contains the declarations of types that are mentioned in the INTYPE file but whose header files are not specified there.

This parameter is required unless the header file for each type is specified individually in the INTYPE file. This parameter is also required if a type not mentioned in the INTYPE file must be generated because other types require it, and these other types are declared in two or more different files.

If the filename specified for the HFILE parameter on the command line or in the INTYPE file does not include an extension, a platform-specific extension such as .H or .h is added automatically.

INTYPE

This parameter specifies the name of the file from which to read the list of object type specifications. The OTT utility translates each type in the list. If the INTYPE parameter is not specified, all types in the user's schema will be translated.

If the filename specified for the INTYPE parameter on the command line does not include an extension, a platform-specific extension such as .TYP or .typ is automatically added.

INTYPE= may be omitted if USERID and INTYPE are the first two parameters, in that order, and USERID= is omitted.

The INTYPE file can be thought of as a makefile for type declarations. It lists the types for which C++ classes are needed.

See Also: "Structure of the INTYPE File" on page 6-18 for more information about the format of the `INTYPE` file

MAPFILE

This parameter specifies the name of the mapping file (*`XXX`*`.cpp`) and corresponding header file (*`XXX`*`.h`) that are generated by the OTT utility. The *`XXX`*`.cpp` file contains the implementation of the functions to register the mappings, while the *`XXX`*`.h` file contains the prototype for the function.

This parameter may be specified either on the command line or in the `INTYPE` file.

MAPFUNC

This parameter specifies the name of the function to be used to register the mappings generated by the OTT utility.

If this parameter is omitted, then the name of the function to register the mappings is derived from the filename specified in the `MAPFILE` parameter.

This parameter may be specified either on the command line or in the `INTYPE` file.

OUTTYPE

This parameter specifies the name of the file into which the OTT utility writes type information for all the object datatypes 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 need to 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 such as `.TYP` or `.typ` is automatically added.

SCHEMA_NAMES

This parameter offers control in qualifying the database name of a type from the default schema that is named in the OUTTYPE file. The OUTTYPE file generated by the OTT utility contains information about the types processed by the OTT utility, including the type names. Valid values include:

- `ALWAYS` (default) -- All type names in the OUTTYPE file are qualified with a schema name.
- `IF_NEEDED` -- 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` -- 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 6–4 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
(
   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.

TRANSITIVE

This parameter indicates whether type dependencies not explicitly listed in the `INTYPE` file are to be translated. Valid values are:

- `TRUE` (default): types needed by other types and not mentioned in the `INTYPE` file are generated

- `FALSE`: types not mentioned in the `INTYPE` file are not generated, even if they are used as attribute types of other generated types.

UNICODE

This parameter specifies whether the application provides unicode (UTF16) support.

- `NONE` (default) --
- `ALL` -- All `CHAR` (`CHAR/VARCHAR`) and `NCHAR` (`NCHAR/NVARCHAR2`) type attributes are declared as `UString` type in the OTT generated C++ class files. The corresponding `get`*`XXX`*`()/set`*`XXX`*`()` return values or parameters are `UString` types. The generated persistent operator new would also take only `UString` arguments.

> **Note:** This setting should be used when both the client characterset and the national characterset is UTF16.

- `ONLYNCHAR` -- Similar to the ALL option, but only NCHAR type attributes will be declared as `UString`.

> **Note:** This setting should be used when the application sets only the Environment's national characterset to UTF16.

Example 6–5 How to Define a Schema for Unicode Support in OTT

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

create type Country as object
(  CNo Number(10),
   CName Varchar2(100),
   CNationalName NVarchar2(100),
   MainCities CitiesList);
```

Example 6–6 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 6–7   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);
...
}
```

USE_MARKER

This parameter indicates whether to support OTT markers for carrying forward user added code. Valid values are:

- `FALSE` (default) -- user added code will not be carried forward, even if the code is added between `OTT_USERCODE_START` and `OTT_USERCODE_END` markers.
- `TRUE` -- code added between the markers `OTT_USER_CODESTART` and `OTT_USERCODE_END` will be carried forward when the same file is generated again.

USERID

This parameter specifies the Oracle username, password, and optional database name (Oracle Net database specification string). If the database name is omitted, the default database is assumed.

```
USERID=username/password[@db_name]
```

If this is the first parameter, then `USERID=` may be omitted as shown:

```
OTT username/password ...
```

This parameter is optional. If omitted, the OTT utility automatically attempts to connect to the default database as user *`OPS$username`*, where *`username`* is the user's operating system username.

Where OTT Parameters Can Appear

Supply OTT parameters on the command line, in a `CONFIG` file named on the command line, or both. Some parameters are also allowed in the `INTYPE` file.

The OTT utility is invoked as follows:

```
OTT parameters
```

You can name a configuration file on the command line with the `CONFIG` parameter as follows:

```
CONFIG=filename
```

If you name this parameter on the command line, then additional parameters are read from the configuration file named *filename*.

In addition, parameters are also read from a default configuration file that resides in an operating system-dependent location. This file must exist, but can be empty. If you choose to enter data in the configuration file, note that no white space is allowed on a line and parameters must be entered one to a line.

If the OTT utility is executed without any arguments, then an online parameter reference is displayed.

The types for the OTT utility to translate are named in the file specified by the INTYPE parameter. The parameters CASE, CPPFILE, HFILE, INITFILE, INITFUNC, MAPFILE, and 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 generated or the MAPFILE and MAPFUNC parameters if C++ codes was generated. The OUTTYPE file, as well as the CPPFILE for C++, specifies the HFILE individually for each type.

The case of the OTT command is operating system-dependent.

File Name Comparison Restriction

Currently, the OTT utility determines if two files are the same by comparing the filenames provided by the user either on the command line or in the INTYPE file. But one potential problem can occur when the OTT utility needs to 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 will be included. If foo1.h and /private/smith/foo1.h are the same file, then a redundant #include will be 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 exactly the same filename.

Using the INTYPE File

When you run the OTT utility, the INTYPE file tells the OTT utility which database types should be translated. The INTYPE file also controls the naming of the generated structures or classes. You can either create an INTYPE file or use the OUTTYPE file of a previous invocation of the OTT utility. If you do not use an INTYPE file, then all types in the schema to which the OTT utility connects are translated.

Overview of the INTYPE File

Example 6–8 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.
- The TRANSLATE ... AS keywords specify that the name of an object attribute should be changed when the type is translated into a C structure. In this case, the SALARY$ attribute of the employee type is translated to salary.
- The AS keyword in the final line specifies that the name of an object type should be changed when it is translated into a structure. In this case, the purchase_order database type is translated into a structure called p_o.

The OTT utility may need 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.

Note: To specify that the OTT utility should not generate required object types that are not specified in the INTYPE file, set TRANSITIVE=FALSE. 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 will report a syntax error in the INTYPE file.

See Also:

- "Structure of the INTYPE File" on page 6-18 for a more detailed specification of the structure of the INTYPE file and the available options.
- "CASE" on page 6-7 for further information regarding the CASE parameter

Structure of the INTYPE File

The INTYPE and OUTTYPE files list the types translated by the OTT utility and provide all the information needed to determine how a type or attribute name is translated to a legal C or C++ identifier. These files contain one or more type specifications. These files also may contain specifications of the following options:

- CASE
- CPPFILE

- HFILE
- INITFILE
- INITFUNC
- MAPFILE
- MAPFUNC

If the CASE, INITFILE, INITFUNC, MAPFILE, or MAPFUNC options are present, then they must precede any type specifications. If these options appear both on the command line and in the INTYPE file, then the value on the command line is used.

> **See Also:** "Overview of the OUTTYPE File" on page 6-31 for an example of a simple user-defined INTYPE file and of the full OUTTYPE file that the OTT utility generates from it

INTYPE File Type Specifications

A type specification in the INTYPE file names an object datatype 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 datatype, and *type_name* is the name of the type. The default schema, if one is not specified, is that of the userID invoking the OTT utility. To use a specific schema, you must use schema_name.

The components of the type specification are:

- *type_name*: Name of the object datatype.
- *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 datatype 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 datatype may need to 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 required by 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 required by types declared in two or more different files, then the translation of the required type is written to the global HFILE file.

Note: 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 will not to generate required object types. The default is TRANSITIVE=TRUE.

Nested #include File Generation

HFILE files generated by the OTT utility #include other necessary files, and #define a symbol constructed from the name of the file. This symbol #defined can then be used to determine if the related HFILE file has already 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
   hfile tott95a.h
type px3
   hfile tott95b.h
```

You invoke the OTT utility as follows:

```
ott scott/tiger intype=tott95i.typ outtype=tott95o.typ code=cpp
```

The OTT utility then generates the following two header files, named tott95a.h and tott95b.h. They are listed in

Example 6–9 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,
      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 6–10   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
```

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. To accomplish this, you would use 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 more than one file, 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 will thereafter follow 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 (/)) should be used in the INTYPE file to specify the location of tott95a.h.

OTT Utility Datatype 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 datatypes of the attributes are mapped to types that are used in Oracle object data types. The datatypes 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 datatypes (for example, BLOB or CLOB).

Example 6–11 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
```

Table 6–2 lists the mappings from types that can be used as attributes to object datatypes that are generated by the OTT utility.

Table 6–2 C++ Object Datatype 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>`

Example 6–12 How to Map Object Datatypes 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 should already exists, and include 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,
         const OCCI_STD_NAMESPACE::string& table);
      void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
                          unsigned int &schemaNameLen, void **typeName,
                          unsigned int &typeNameLen) const;
      object_type();
      object_type(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
      static void *readSQL(void *ctxOCCI_);
      virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
      static void writeSQL(void *objOCCI_, void *ctxOCCI_);
      virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
};

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

class other_type : public oracle::occi::PObject
{
   protected:
      oracle::occi::Number object_number;

   public:
      void *operator new(size_t size);
      void *operator new(size_t size, const oracle::occi::Connection * sess,
         const OCCI_STD_NAMESPACE::string& table);
      void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
                          unsigned int &schemaNameLen, void **typeName,
                          unsigned int &typeNameLen) const;
      other_type();
      other_type(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
      static void *readSQL(void *ctxOCCI_);
      virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
      static void writeSQL(void *objOCCI_, void *ctxOCCI_);
      virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
};

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

class many_types : public oracle::occi::PObject
{
   protected:
      OCCI_STD_NAMESPACE::string the_varchar;
      OCCI_STD_NAMESPACE::string the_char;
      oracle::occi::Blob the_blob;
      oracle::occi::Clob the_clob;
      object_type * the_object;
      oracle::occi::Ref< other_type > another_ref;
      oracle::occi::Ref< many_types > the_ref;
      OCCI_STD_NAMESPACE::vector< oracle::occi::Number > the_varray;
      OCCI_STD_NAMESPACE::vector< object_type * > the_table;
      oracle::occi::Date the_date;
      oracle::occi::Number the_num;
      oracle::occi::Bytes the_raw;

   public:
      void *operator new(size_t size);
      void *operator new(size_t size, const oracle::occi::Connection * sess,
         const OCCI_STD_NAMESPACE::string& table);
      void getSQLTypeName(oracle::occi::Environment *env, void **schemaName,
                          unsigned int &schemaNameLen, void **typeName,
                          unsigned int &typeNameLen) const;
      many_types();
      many_types(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
      static void *readSQL(void *ctxOCCI_);
      virtual void readSQL(oracle::occi::AnyData& streamOCCI_);
      static void writeSQL(void *objOCCI_, void *ctxOCCI_);
      virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);
};

#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.

Default Name Mapping

When the OTT utility creates a C or C++ identifier name for an object type or attribute, it translates the name from the database character set to a legal C or C++ identifier. First, the name is translated from the database character set to the character set used by the OTT utility. Next, if a translation of the resulting name is supplied in the INTYPE file, that translation is used. Otherwise, the OTT utility translates the name character-by-character to the compiler character set, applying the character case specified in the CASE parameter. The following text describes this in more detail.

When the OTT utility reads the name of a database entity, the name is automatically translated from the database character set to the character set used by the OTT utility. In order for the OTT utility to read the name of the database entity successfully, all the characters of the name must be found in the OTT character set, although a character may have different encodings in the two character sets.

The easiest way to guarantee that the character set used by the OTT utility contains all the necessary characters is to make it the same as the database character set. Note, however, that the OTT character set must be a superset of the compiler character set. That is, if the compiler character set is 7-bit ASCII, then the OTT character set must include 7-bit ASCII as a subset, and if the compiler character set is 7-bit EBCDIC, then the OTT character set must include 7-bit EBCDIC as a subset. The user specifies the character set that the OTT utility uses by setting the NLS_LANG environment variable, or by some other operating system-specific mechanism.

Once the OTT utility has read the name of a database entity, it translates the name from the character set used by the OTT utility to the compiler's character set. If a translation of the name appears in the INTYPE file, then the OTT utility uses that translation.

Otherwise, the OTT utility attempts to translate the name as follows:

1. If the OTT character set is a multibyte character set, all multibyte characters in the name that have single-byte equivalents are converted to those single-byte equivalents.

2. The name is converted from the OTT character set to the compiler character set. The compiler character set is a single-byte character set such as US7ASCII.
3. The case of letters is set according to how the `CASE` parameter is defined, and any character that is not legal in a C or C++ identifier, or that has no translation in the compiler character set, is replaced by an underscore character (_). If at least one character is replaced by an underscore, then the OTT utility gives a warning message. If all the characters in a name are replaced by underscores, the OTT utility gives an error message.

Character-by-character name translation does not alter underscores, digits, or single-byte letters that appear in the compiler character set, so legal C or C++ identifiers are not altered.

Name translation may, for example, translate accented single-byte characters such as *o* with an umlaut or an *a* with an accent grave to *o* or *a*, with no accent, and may translate a multibyte letter to its single-byte equivalent. Name translation will typically fail 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 will not detect a naming clash caused by two or more database identifiers being mapped to the same C name, nor will it detect a naming problem where a database identifier is mapped to a C keyword.

Overview of the OUTTYPE File

The `OUTTYPE` file is named on the OTT command line. When the OTT utility generates a C++ header file, it also writes the results of the translation into the `OUTTYPE` file. This file contains an entry for each of the translated types, including its version string and the header file to which its C++ representation was written.

The `OUTTYPE` file from one OTT utility run can be used as the `INTYPE` file for a subsequent invocation of the OTT utility.

Example 6–13 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
   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`.

The OTT utility analyzes the types in the `INTYPE` file for type dependencies before performing the translation, and translates other types as necessary.

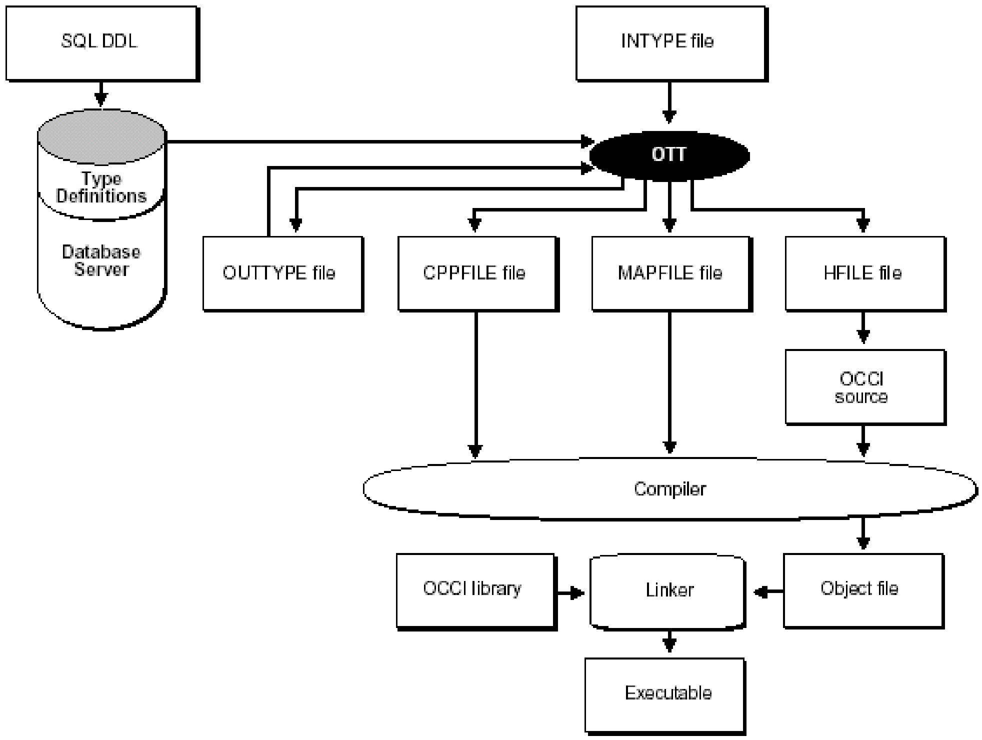
Note: To specify that the OTT utility should not generate required object types that are not specified in the `INTYPE` file, set `TRANSITIVE=FALSE`. The default is `TRANSITIVE=TRUE`.

The OTT Utility and OCCI Applications

The OTT utility generates objects and maps SQL datatypes 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 that 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 6–1 shows the steps involved in using the OTT utility with OCCI. These steps are described following the figure.

Figure 6–1 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 containing the prototype of the function (`MAPFUNC`) that registers the mappings.
- A C++ source file (with the extension `.cpp`) that contains the static methods to be called by OCCI while instantiating the objects. Do not call these methods directly from your OCCI application. The filename is specified on the OTT command line by the `CPPFILE` parameter.
- 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 (the `OUTTYPE` file) that contains an entry for each of the translated types, including the version string and the file into which it is written. The filename is specified on the OTT command line by the `OUTTYPE` parameter.

4. Write the OCCI application and include the header files created by the OTT utility in the OCCI source code file.

   The application declares an environment and calls the function `MAPFUNC` to register the mappings.

5. Compile the OCCI application to create the OCCI object code, and link the object code with the OCCI libraries to create the program executable.

C++ Classes Generated by the OTT Utility

When the OTT utility generates a C++ class from a database object type, the class declaration contains one element corresponding to each attribute of the object type. The datatypes of the attribute are mapped to types that are used in Oracle object datatypes, as defined in Table 6–2 on page 6-26.

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 already 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 6–14 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=scott/tiger intype=demoin_3.typ outype=demoout_3.typ code=cpp
hfile=demo_3.h cppfile=demo_3.cpp
```

Map Registry Function

One function to register the mappings with the environment is generated by the OTT utility. The function contains the mappings for all the types translated by the invocation of the OTT utility. The function name is either specified in the MAPFUNC parameter or, if that parameter is not specified, derived from MAPFILE parameter. The only argument to the function is the pointer to Environment.

The function uses the provided Environment to get Map and then registers the mapping of each translated type.

Extending C++ Classes

To enhance the functionality of a class generated by the OTT utility, you can derive new classes. You can also add methods to a class, but Oracle does not recommend doing so due to an inherent risk.

> **See Also:** "Carrying Forward User Added Code" on page 6-37 for details on how to use OTT markers to retain code you want to add in OTT generated files

Assume that you want to generate the both `CAddress` and `MyAddress` classes from the SQL object type `ADDRESS`. `MyAddress` class can be derived from `CAddress` class. To do this, the OTT utility must alter the code it generates:

- 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.

> **Caution:** 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 6–15 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:

```
CASE = SAME
MAPFILE = RegisterMappings_5.cpp

TYPE FULL_NAME GENERATE CFullName AS MyFullName
```

```
   TRANSLATE first_name as FirstName
             last_name as LastName

TYPE ADDRESS GENERATE CAddress AS MyAddress
TYPE PERSON GENERATE CPerson AS MyPerson
TYPE STUDENT GENERATE CStudent AS MyStudent
```

Carrying Forward User Added Code

To extend the functionality of OTT generated code, at times programmers may want to add code in the OTT generated file. The way OTT can distinguish between OTT generated code and code added by the user is by looking for some predefined markers (tags). OTT recognizes `OTT_USERCODE_START` as the "start of user code marker", and `OTT_USERCODE_END` as the "end of user code marker".

For OTT marker support, a user block is defined as

```
OTT_USERCODE_START + user added code + OTT_USERCODE_END
```

OTT marker support enables carrying forward the user added blocks in *.h and *.cpp files.

Properties of OTT Markers

These items describe the properties of OTT Markers Support:

1. User must use the command line option `USE_MARKER=TRUE` from the very first time OTT is invoked to generate a file.
2. User should treat markers like other C++ statements; a marker will be 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 will copy the text/code given within markers verbatim along with the markers while generating the code next time,

User modified code:

```
// --- modified generated code
OTT_USERCODE_START
// --- including "myfullname.h"
#ifndef MYFULLNAME_ORACLE
#include "myfullname.h"
#endif
OTT_USERCODE_END
// --- end of code addition
```

Carried forward code:

```
OTT_USERCODE_START
// --- including "myfullname.h"
#ifndef MYFULLNAME_ORACLE
#include "myfullname.h"
#endif
OTT_USERCODE_END
```

5. OTT will not be able to 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 will fail to find out the class name with which the code was associated earlier as the case of the class name got modified by the user in the `INTYPE` file.

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

   - If user asks to generate the class with different name (`GENERATE AS` clause of `INTYPE` file), OTT will fail to find out the class name with which the code was associated earlier as the class name got modified by the user in the `INTYPE` file.

```
CASE=LOWER                           CASE=LOWER
TYPE employee                        TYPE employee
TRANSLATE SALARY$ AS salary          TRANSLATE SALARY$ AS salary
   DEPTNO AS department                 DEPTNO AS department
TYPE ADDRESS                         TYPE ADDRESS
TYPE item                              TYPE item
TYPE "Person"                          TYPE "Person"
TYPE PURCHASE_ORDER AS p_o             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 will flag an error if:
   - it does not find a matching `OTT_USERCODE_END` for `OTT_USERCODE_START` encountered
   - markers are nested (OTT finds next `OTT_USERCODE_START` before `OTT_USERCODE_END` is found for the previous `OTT_USERCODE_START`)
   - `OTT_USERCODE_END` is encountered before `OTT_USERCODE_START`

Using OTT Markers

The user must use command line option `USE_MARKER=TRUE` to turn on marker support. There are two general ways in which OTT markers can carry forward user added code:

1. **User code added in .h file.**
   - **User code added in global scope.** This is typically the case when user needs to 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 will have private scope for data members and public scope for methods, or protected scope for data members and public scope for methods. User blocks can be added after all OTT generated declarations in either access specifiers.

Example 6–16 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
...
```

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

2. **User code added in .cpp file.** OTT will support 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 more than one 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 will not be carried forward.

Example 6–17 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
```

7 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

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, username/passwords, error messages, object names, and so on) have been extended to handle data in any characterset. Character data from relational tables or objects can be in any characterset. OCCI can be used to develop multi-lingual, global and Unicode applications.

Specifying Charactersets

OCCI applications need to 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` datatype (`CHAR`, `VARCHAR2`, `LONG`) columns/attributes. The client national characterset specifies the characterset for data of all `NCHAR` datatype (`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.

Example 7–1 How to Use Globalization and Unicode Support

```
Environment *env = Environment:createEnvironment("JA16SJIS","UTF8");
```

This statement creates a OCCI Environment with JA16SJIS as the client characterset and UTF8 as the client national characterset.

Any valid Oracle characterset name (except 'AL16UTF16') can be passed to createEnvironment(). A 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");
```

Note: 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.

Datatypes for Globalization and Unicode Support

The datatypes described in this section include:

- UString Datatype

UString Datatype

OCCI has introduced a new datatype: UString, to enable 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` datatype.

```
typedef basic_string<utext> UString;
```

Oracle's `utext` datatype is a 2 byte short datatype 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` datatype. This is because the `wchar_t` datatype is type defined to a 2 byte `short` in these platforms, which is same as Oracle's `utext`, allowing applications to use a `wstring` type variable where a `UString` would be normally required. Consequently, applications can also pass wide-character string literals, created by prefixing the literal with the letter 'L', to OCCI Unicode APIs.

Example 7–2 Using wstring Datatype

```
//bind Unicode data using wstring datatype
//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 = env->createConnection(L"SCOTT",L"TIGER",L"");
```

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

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`.

Example 7–3 Binding UTF8 Data Using the string Datatype

```
//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 datatype: UString and the OCCI UTF16 interfaces should be used.

Example 7–4 Binding UTF16 Data Using the UString Datatype

```
//bind Unicode data using UString datatype
//binding the Euro symbol, UTF16 codepoint 0x20AC
utext eurochars[] = {0x20AC,0x00};
UString eurostr(eurochars);
stmt->setUString(1,eurostr);//use the UString interface
```

CLOB and NCLOB Datatypes

Oracle provides the `CLOB` and `NCLOB` datatypes for storing and processing large amounts of character data. `CLOB`s represent data in the database characterset and `NCLOB`s represent data in the database national characterset. `CLOB`s and `NCLOB`s 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` datatypes. If the database type is `NCLOB`, then the Clob setCharSetForm() 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 `CLOB`s and `NCLOB`s). 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.

Example 7–5 Using CLOB and NCLOB Datatypes

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

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`.

Objects and OTT Support

Multibyte and UTF16 charactersets are supported for handling character data in object attributes. All `CHAR` datatype (`CHAR/VARCHAR2`) attributes hold data in the client-side characterset, while all `NCHAR` datatype (`NCHAR/NVARCHAR2`) attributes hold data in the client-side national characterset. A member variable of `UString` datatype represents an attribute in UTF16 characterset.

See Also:

- Chapter 10, "OCCI Application Programming Interface": two new versions of operator new() on page 10-198 that have been added to the PObject Class for object support
- Chapter 6, "Object Type Translator Utility": a new UNICODE parameter on page 6-13 that has been added for OTT utility support.

8 Oracle Streams Advanced Queuing

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

This chapter contains these topics:

See Also:

- *Oracle Streams Advanced Queuing User's Guide and Reference* for basic concepts of Advanced Queuing
- Chapter 10, "OCCI Application Programming Interface"

Overview of Oracle Streams Advanced Queuing

Oracle Streams is a new information sharing feature that provides replication, message queuing, data warehouse loading, and event notification. It is also the foundation behind Oracle Streams Advanced Queuing (AQ).

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

- Perform message queuing operations similar to SQL operations from the Oracle database
- Communicate asynchronously through messages in AQ queues
- Integrate with database for unprecedented levels of operational simplicity, reliability, and security to message queuing
- Audit and track messages
- Supports both synchronous and asynchronous modes of communication

See Also: http://otn.oracle.com/products/dataint/index.html for more information about the Advanced Queuing feature

The advantages of using AQ in OCCI applications include:

- Create applications that communicate with each other in a consistent, reliable, secure, and autonomous manner
- Store messages in database tables, bringing the reliability and recoverability of the database to your messaging infrastructure
- Retain messages in the database automatically for auditing and business intelligence
- Create applications that leverage messaging without having to deal with a different security, data type, or operational mode
- Leverage transactional characteristics of the database

Since traditional messaging solutions have single subscriber queues, a queue must be created for each pair of applications that communicate with each other. The publish/subscribe protocol of the AQ makes it easy to add additional applications (subscribers) to a conversation between multiple applications.

AQ Implementation in OCCI

OCCI AQ is a set of interfaces that allows messaging clients to access the Advanced Queuing feature of Oracle for enterprise messaging applications. Currently, OCCI AQ supports only the operational interfaces and not the administrative interface, but administrative operations can be accessed through embedded PL/SQL calls.

See Also: Package DBMS_AQADM in *PL/SQL Packages and Types Reference* for administrative operations in AQ support through PL/SQL

The AQ feature can be used in conjunction 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

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.

> **See Also:** Message Class documentation in Chapter 10, "OCCI Application Programming Interface"

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 8–1 demonstrates an instantiation of a new Agent object in a client program.

Example 8–1 Creating an Agent

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

See Also: Agent Class documentation in Chapter 10, "OCCI Application Programming Interface"

Producer

A client uses a Producer object to enqueue Messages into a queue. It is also used to specify various enqueue options.

See Also: Producer Class documentation in Chapter 10, "OCCI Application Programming Interface"

Consumer

A client uses a Consumer object to dequeue Messages that have been delivered to a queue. It also specifies various dequeuing options.

See Also: Consumer Class documentation in Chapter 10, "OCCI Application Programming Interface"

Listener

A Listener listens for Messages for registered Agents at specified queues.

See Also: Listener Class documentation in Chapter 10, "OCCI Application Programming Interface"

Subscription

A Subscription encapsulates the information and operations necessary for registeringa subscriber for notifications.

Creating Messages

As mentioned previously, a `Message` is a basic unit of information that contains both the properties of the message and its content, or ***payload***. Each message is enqueued by the `Producer` and dequeued by the `Consumer` objects.

Message Payloads

OCCI supports three types of message payloads:

- RAW
- AnyData
- User-defined

RAW

`RAW` payloads are mapped as objects of the Bytes Class in OCCI.

AnyData

The `AnyData` type models self-descriptive data encapsulation; it contains both the type information and the actual data value. Data values of most SQL types can be converted to `AnyData`, and then be converted to the original data type. `AnyData` also supports user-defined data types. The advantage of using `AnyData` payloads is that it ensures both type preservation after an enqueue and dequeue process, and that it allows the user to use a single queue for all types used in the application. Example 8–2 demonstrates how to create an `AnyData` message. Example 8–3 shows how to retrieve the original data type from the message.

Example 8–2 Creating an AnyData Message with a String Payload

```
AnyData any(conn);
any.setFromString("item1");
Message mes(env);
mes.setAnyData(any);
```

Example 8–3 Determining the Type of the Payload in an AnyData Message

```
TypeCode tc = any.getType();
```

User-defined

OCCI supports enqueuing and dequeuing of user-defined types as payloads. Example 8–4 demonstrates how to create a payload with a user-defined `Employee` object.

Example 8–4 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);
```

Message Properties

Aside from payloads, the user can specify several additional message properties:

- Correlation
- Sender
- Delay and Expiration
- Recipients
- Priority and Ordering

Correlation

Applications can specify a correlation identifier of the message during the enqueuing process, as demonstrated in Example 8–5. This identifier can then be used by the dequeuing application.

Example 8–5 Specifying the Correlation identifier

```
mes.setCorrelationId("enq_corr_di");
```

Sender

Applications can specify the sender of the message, as demonstrated in Example 8–6. The sender identifier can then be used by the receiver of the message.

Example 8–6 Specifying the Sender identifier

```
mes.setSenderId(agt);
```

Delay and Expiration

Time settings control the delay and expiration times of the message in seconds, as demonstrated in Example 8–7.

Example 8–7 Specifying the Delay and Expiration times of the message

```
mes.setDelay(10);
mes.setExpirationTime(60);
```

Recipients

The agents for whom the message is intended can be specified during message encoding, as demonstrated in Example 8–8. This ensures that only the specified recipients can access the message.

Example 8–8 Specifying message recipients

```
vector<Agent> agt_list;
for (i=0; i<num_recipients; i++)
   agt_list.push_back(Agent(name, address, protocol));
mes.setRecipientList(agt_list);
```

Priority and Ordering

By assigning a priority level to a message, the sender can control the order in which the messages are dequeued by the receiver. Example 8–9 demonstrates how to set the priority of a message.

Example 8–9 Specifying the priority of a message

```
mes.setPriority(3);
```

Enqueuing Messages

Messages are enqueued by the Producer. The Producer Class is also used to specify enqueue options. A `Producer` object can be created on a valid connection where enqueuing will be performed, as illustrated in Example 8–10.

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 8–10, or only after the enclosing transaction has been committed.

To enqueue the message, use the send() method, as demonstrated in Example 8–10. A client may retain the `Message` object after it is sent, modify it, and send it again.

Example 8–10 Creating a Producer, setting visibility, and enqueuing the message

```
Producer prod(conn);
...
prod.setVisibility(Producer::ENQ_IMMEDIATE);
...
prod.send(mes, queueName);
```

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 8–11.

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 8–11.

To dequeue the message, use the receive() method, as demonstrated in Example 8–11. The `typeName` and `schemaName` parameters of the receive() method specify the type of payload and the schema of the payload type.

Example 8–11 Creating a Consumer, Naming the Consumer, and Receiving a Message

```
Consumer cons(conn);
...
cons.setConsumerName("BillApp");
cons.setQueueName(queueName);
```

```
...
Message mes = cons.receive(Message::OBJECT, "BILL_TYPE", "BILL_PROCESSOR");
```

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 8–12.

Example 8–12 Receiving a Message

```
//receiving a RAW message
Message mes = cons.receive(Message::RAW);
...
//receiving an ANYDATA message
Message mes = cons.receive(Message::ANYDATA);
...
```

Dequeuing Options

The dequeuing application can specify several dequeuing options before it begins to receive messages. These include:

- Correlation
- Mode
- Navigation

Correlation

The message can be dequeued based on the value of its correlation identifier using the setCorrelationId() method, as shown in Example 8–13.

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 8–13.

Navigation

Messages enqueued in a single transaction can be viewed as a single group by implementing the setPositionOfMessage() method, as shown in Example 8–13.

Example 8–13 Specifying dequeuing options

```
cons.setCorrelationId(corrId);
...
cons.setDequeueMode(deqMode);
...
cons.setPositionOfMessage(Consumer::DEQ_NEXT_TRANSACTION);
```

Listening for Messages

The Listener listens for messages on queues on behalf of its registered clients. The Listener Class implements the listen() method, which is a blocking call that returns once a queue has a message for one of the registered agents, or throws an error when the time out period expires. Example 8–14 illustrates the listening protocol.

Example 8–14 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;
}
```

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 need to be connected to a database.

An OCCI application can do all of the following:

- Register interest in notification in the AQ namespace, and be notified when an enqueue occurs.
- Register interest in subscriptions to database events, and receive notifications when these events are triggered.
- Manage registrations, such as disable registrations temporarily, or dropping registrations entirely.
- Post (or send) notifications to registered clients.

Publish-Subscribe Notifications

Notifications can work in several ways. They can be:

- received directly by the OCCI application
- sent to a pre-specified e-mail address
- sent to a pre-defined HTTP URL
- invoke a pre-specified database PL/SQL procedure

Registered clients are notified asynchronously when events are triggered, or on an explicit AQ enqueue. Clients do not need to be connected to a database for notifications to work. Registration can be accomplished in two ways:

- Direct Registration
- Open Registration

Direct Registration

You can register directly with the database. This is relatively simple, and the registration takes effect immediately. Example 8–15 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 8–15 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:

Namespace To receive notifications from AQ queues, the namespace must be set to `Subscription::NS_AQ`. To receive notifications from other applications that use `conn->postToSubscription()` method, the namespace must be set to `Subscription::NS_ANONYMOUS`.

Protocol The protocol can be set to these options:

- If an OCCI client needs to receive an event notification, this attribute should be set to `Subscription::PROTO_CBK`. You also need to set the notification callback and the subscription context before registering the `Subscription`. The notification callback will be called when the event occurs.
- For an e-mail notification, set the protocol to `Subscription::PROTO_MAIL`. You must set the recipient name prior to subscribing to avoid an application error.
- For an HTTP URL notification, set the protocol to `Subscription::HTTP`. You must set the recipient name prior to 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 prior to subscribing to avoid an application error.

4. Register the subscriptions using `connection->registerSubscriptions()`.

Open Registration

You can also register through an intermediate LDAP that sends the registration request to the database. This is used when the client cannot have a direct database connection; for example, the client wants to register for an open event while the database is down. This approach is also used when a client wants to register for the same event(s) in multiple databases, concurrently.

Example 8–16 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 role `ENTERPRISE_AQ_USER_ROLE`.

  - For each enterprise domain, add enterprise role `ENTERPRISE_AQ_USER_ROLE` to 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 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
```

Example 8–16 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:

   Namespace To receive notifications from AQ queues, the namespace must be set to `Subscription::NS_ANONYMOUS`. To receive notifications from other applications that use `conn->postToSubscription()` method, the namespace must be set to `Subscription::NS_ANONYMOUS`.

Protocol The protocol can be set to these options:

- If an OCCI client needs to receive an event notification, this attribute should be set to `Subscription::PROTO_CBK`. You also need to set the notification callback and the subscription context before registering the `Subscription`. The notification callback will be 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 will 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 will be posted.
- To invoke PL/SQL procedures in the database on event notification, set protocol to `Subscription::PROTO_SERVER`. You must set the recipient name to the database procedure invoked on notification.

6. Register the subscription: `environment->registerSubscriptions()`.

Open registration will take effect when the database accesses LDAP to pick up new registrations. The frequency of pick-ups is determined by the value of `REG_SYNC_INTERVAL`.

Clients can temporarily disable subscriptions, re-enable them, or permanently unregister from future notifications.

Notification Callback

The client needs to register a notification callback. This callback is invoked only when there is some activity on the registered subscription. In the Streams AQ namespace, this happens when a message of interest is enqueued.

The callback must return `0`, and it must have this specification:

```
typedef unsigned int (*callbackfn) (Subscription &sub, NotifyResult *nr);
```

where:

- `sub` - `Subscription` object which was used when the callback was registered.
- `nr - NotifyResult` object holding the notification info.

Note: 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 8–1. Only a bytes payload is currently supported, and you must explicitly dequeue messages rom 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 8–1 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

Message Format Transformation

Applications often use data in different formats, and this requires a type transformation. A transformation is implemented as a SQL function that takes the source data type as input and returns an object of the target data type.

Transformations can be applied when message are enqueued, dequeued, or when they are propagated to a remote subscriber.

See Also: The following chapters of the *Oracle Streams Advanced Queuing User's Guide and Reference* for information of format transformation

- Oracle Streams AQ Administrative Interface
- Oracle Streams AQ Administrative Interface: Views
- Oracle Streams AQ Operational Interface: Basic Operations

9 Oracle XA Library

The Oracle XA library is an external interface that allows transaction managers other than the Oracle server to coordinate global transactions. XA library use supports non-Oracle resource managers, in distributed transactions. This is particularly useful in transactions between several databases and resources.

The implementation of the Oracle XA library conforms to the X/Open Distributed Transaction Processing (DTP) software architecture's XA interface specification. The Oracle XA Library is installed as part of the Oracle Database Enterprise Edition.

This chapter contains these topics:

- Application Development with XA and OCCI
- APIs for XA Support

See Also:

- http://www.opengroup.org
- *Oracle Database Application Developer's Guide - Fundamentals* for more details on the Oracle XA library and architecture
- Chapter 10, "OCCI Application Programming Interface"

Application Development with XA and OCCI

For connection, disconnection, and transaction control on Oracle databases, applications must interface with a transaction manager. OCCI has APIs for interacting with `Environment` and `Connection` objects within XA and make them available for Oracle database access, such as `SELECT` queries, DML statements, object access, and so on.

Example 9–1 How to Use Transaction Managers with XA

```
/* Transaction manager opens connection to the Oracle server*/
tpopen("oracle_xa+acc=p/SCOTT/TIGER+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/SCOTT/TIGER+sestm=10");
Connection *xaconn = xaenv->getXAConnection(
   "oracle_xa+acc=p/SCOTT/TIGER+sestm=10");

/* Use the Environment & Connection objects */
Statement *stmt = xaconn->createStatement(
   "Update Emp set sal = sal * 0.2");

...

/* Release the Environment & Connection objects */
xaenv->releaseXAConnection(xaconn);
Environment::releaseXAEnvironment(xaenv);
```

APIs for XA Support

The following methods of the Environment Class support use of XA libraries:

- getXAConnection() on page 10-111
- getXAEnvironment() on page 10-112
- releaseXAConnection() on page 10-112
- releaseXAEnvironment() on page 10-112

In addition, the getXAErrorCode() method on page 10-247 should be used by XA enabled applications to determine if thrown exceptions are due to an SQL error (`XA_OK`) or an XA error (an XA error code).

10 OCCI Application Programming Interface

This chapter describes the OCCI classes and methods for C++.

See Also:

- Format Models in *Oracle Database SQL Reference*
- Table A-1 in *Oracle Database Globalization Support Guide*

OCCI Classes and Methods

Table 10–1 provides a brief description of all the OCCI classes. This section is followed by detailed descriptions of each class and its methods.

Table 10–1 Summary of OCCI Classes

Class	Description
Agent Class on page 10-10	Represents an agent in the Advanced Queuing context.
AnyData Class on page 10-14	Provides methods for the Object Type Translator (OTT) utility, read/write SQL methods for linearization of objects, and conversions to and from other datatypes.
Bfile Class on page 10-26	Provides access to a SQL `BFILE` value.
Blob Class on page 10-35	Provides access to a SQL `BLOB` value.
Bytes Class on page 10-45	Examines individual bytes of a sequence for comparing bytes, searching bytes, and extracting bytes.
Clob Class on page 10-48	Provides access to a SQL `CLOB` value.
Connection Class on page 10-61	Represents a connection with a specific database.
ConnectionPool Class on page 10-72	Represents a connection pool with a specific database.
Consumer Class on page 10-78	Supports dequeuing of `Messages` and controls the dequeuing options.
Date Class on page 10-89	Specifies abstraction for SQL DATE data items. Also provides formatting and parsing operations to support the OCCI escape syntax for date values.
Environment Class on page 10-102	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 on page 10-118	Represents a period of time in terms of days, hours, minutes, and seconds.
IntervalYM Class on page 10-131	Represents a period of time in terms of year and months.
Listener Class on page 10-142	Listens on behalf of one or more agents on one or more queues.
Map Class on page 10-145	Used to store the mapping of the SQL structured type to C++ classes.

Table 10–1 (Cont.) Summary of OCCI Classes

Class	Description
Message Class on page 10-147	A unit that is enqueued or dequeued.
MetaData Class on page 10-158	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 on page 10-166	Used to hold notification information from the Streams AQ callback function.
Number Class on page 10-168	Models the numerical datatype.
PObject Class on page 10-194	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 on page 10-201	Supports enqueuing options and enqueues `Messages`.
Ref Class on page 10-207	The mapping in C++ for the SQL REF value, which is a reference to a SQL structured type value in the database.
RefAny Class on page 10-214	The mapping in C++ for the SQL REF value, which is a reference to a SQL structured type value in the database.
ResultSet Class on page 10-219	Provides access to a table of data generated by executing an OCCI `Statement`.
SQLException Class on page 10-245	Provides information on database access errors.
StatelessConnectionPool Class on page 10-249	Represents a pool of stateless, authenticated connections to the database.
Statement Class on page 10-260	Used for executing SQL statements, including both query statements and insert / update / delete statements.
Stream Class on page 10-316	Used to provide streamed data (usually of the LONG datatype) to a prepared DML statement or stored procedure call.
Subscription Class on page 10-319	Encapsulates the information and operations necessary for registering a subscriber for notification.

Table 10–1 (Cont.) Summary of OCCI Classes

Class	Description
Timestamp Class on page 10-327	Specifies abstraction for SQL TIMESTAMP data items. Also provides formatting and parsing operations to support the OCCI escape syntax for time stamp values.

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.

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.

OCCI Support for Windows NT

The following global methods are designed for accessing collections of `Refs` in ResultSet Class and Statement Class on Windows NT. While method names changed, the number of parameters and their types remain the same.

- Use `getVectorOfRefs()` in place of `getVector()` on Windows NT.
- Use `setVectorOfRefs()` in place of `setVector()` on Windows NT.

Applications on Windows NT should be calling these new methods only for retrieving and inserting collections of references. Applications not running on Windows NT can use either set of accessors. However, Oracle recommends the use of the new methods for any vector operations with `Refs`.

Working with Collections of Refs

Collections of Refs can be fetched and inserted using methods of the following classes:

ResultSet Class

Fetching Collection of Refs Use the following version of getVectorOfRefs() on page 10-237 to return a column of references:

```
void getVectorOfRefs(
   ResultSet  *rs,
   unsigned int index,
   OCCI_STD_NAMESPACE::vector<Ref<T> > &vect);
```

Statement Class

Fetching Collection of Refs Use getVectorOfRefs() on page 10-283 to return a collection of references from a column:

```
void getVectorOfRefs(
   Statement  *stmt,
   unsigned int index,
   OCCI_STD_NAMESPACE::vector<Ref<T> > &vect);
```

Inserting a Collection of Refs Use setVectorOfRefs() on page 10-313 to insert a collection of references into a column:

```
template  <class T>
void setVectorOfRefs(
   Statement *stmt,
   unsigned int paramIndex,
   const OCCI_STD_NAMESPACE::vector<Ref<T> > &vect,
   const OCCI_STD_NAMESPACE::string &sqltype);
```

Inserting a Collection of Refs: Multibyte Support The following method should be used for multibyte support:

```
void setVectorOfRefs(
   Statement *stmt,
   unsigned int paramIndex,
   const OCCI_STD_NAMESPACE::vector<Ref<T> > &vect,
   const OCCI_STD_NAMESPACE::string &schemaName,
   const OCCI_STD_NAMESPACE::string &typeName);
```

Inserting a Collection of Refs: UString (UTF16) Support The following method should be used for `UString` support:

```
template <class T>
void setVectorOfRefs(
   Statement *stmt,
   unsigned int paramIndex,
   const OCCI_STD_NAMESPACE::vector<Ref<T> > &vect,
   const UString &schemaName,
   const UString &typeName);
```

Working with Collections of Objects

The global methods for the fetching or inserting of collections of objects have been changed for Windows NT. The interface remains the same with respect to the method names and the number of parameters and the datatypes, but differs in the template parameter definition for Windows NT. Specifically, the template parameter for the template methods of `getVector()` and `setVector()` of objects (object pointers) on Windows NT have a `T` instead of a `T*` as shown in the following APIs.

The methods are used in the same way on different operating systems, and you don't need to modify the call to these methods. On Windows NT, the template arguments passed as object pointers in the method call are specialized for parameter `T`, instead of a `T*` on other operating systems.

Collections of objects can be fetched and inserted using methods of the following classes:

ResultSet Class

Fetching a Collection of objects This method fetches a collection of objects from a `ResultSet` for the column specified by the index.

```
#ifdef WIN32COMMON
   template <class T>
   void getVector( ResultSet *rs, unsigned int index,
     OCCI_STD_NAMESPACE::vector< T > &vect);
#else
  template <class T>
    void getVector( ResultSet *rs, unsigned int index,
      OCCI_STD_NAMESPACE::vector< T* > &vect);
#endif
```

Statement Class

Fetching a Collection of Objects This method fetches a collection of objects from a statement for the column specified by the index. This method is used in case of `OUT` binds and DML returning clauses.

```
#ifdef WIN32COMMON
   template <class T>
   void getVector( Statement *stmt, unsigned int index,
     OCCI_STD_NAMESPACE::vector< T > &vect);
  #else
   template <class T>
   void getVector( Statement *stmt, unsigned int index,
     OCCI_STD_NAMESPACE::vector< T* > &vect);
  #endif
```

Inserting a Vector of Objects This method inserts a collection of objects into a statement for the column specified by the index.

```
#ifdef WIN32COMMON
  template <class T>
  void setVector( Statement *stmt, unsigned int paramIndex,
    const OCCI_STD_NAMESPACE::vector< T > &vect,
    const  OCCI_STD_NAMESPACE::string &sqltype);
```

```
#else
  template <class T>
  void setVector( Statement *stmt, unsigned int paramIndex,
    const OCCI_STD_NAMESPACE::vector<T* > &vect,
    const OCCI_STD_NAMESPACE::string &sqltype);
#endif
```

Inserting a Vector of Objects: Multibyte Support The following method should be used for multibyte support:

```
#ifdef WIN32COMMON
template <class T>
void setVector(
   Statement *stmt,
   unsigned int paramIndex,
   const OCCI_STD_NAMESPACE::vector< T > &vect,
   const OCCI_STD_NAMESPACE::string &schemaName,
   const OCCI_STD_NAMESPACE::string &typeName);
#else
template <class T>
void setVector(
   Statement *stmt,
   unsigned int paramIndex,
   const OCCI_STD_NAMESPACE::vector< T*> &vect,
   const OCCI_STD_NAMESPACE::string &schemaName,
   const OCCI_STD_NAMESPACE::string &typeName);
#endif
```

Inserting a Collection of Objects: UString (UTF16) Support The following method should be used for `UString` support:

```
#ifdef WIN32COMMON
template <class T>
void setVector(
   Statement *stmt,
   unsigned int paramIndex,
   const OCCI_STD_NAMESPACE::vector< T > &vect,
   const UString &schemaName,
   const UString &typeName);
#else
template <class T>
void setVector(
   Statement *stmt,
   unsigned int paramIndex,
   const OCCI_STD_NAMESPACE::vector< T*> &vect,
   const UString &schemaName,
   const USring &typeName);
#endif
```

Agent Class

The `Agent` class represents an agent in the Advanced Queuing context.

Table 10–2 Summary of Agent Methods

Method	Summary
Agent() on page 10-10	`Agent` class constructor.
getAddress() on page 10-11	Return the address of the `Agent`.
getName() on page 10-11	Return the name of the `Agent`.
getProtocol() on page 10-11	Return the protocol of the `Agent`.
isNull() on page 10-11	Test whether the `Agent` object is `NULL`.
setAddress() on page 10-12	Set address of the `Agent` object.
setName() on page 10-12	Set name of the `Agent` object.
setNull() on page 10-12	Set `Agent` object to `NULL`.
setProtocol() `on page 10-12`	Set protocol of the `Agent` object.

Agent()

Agent class constructor.

Syntax	Description

`Agent(`
`   Environment *env);`
	Creates an `Agent` object initialized to its default values.

`Agent(`
`   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

Parameter	Description
`name`	Name
`address`	Address
`protocol`	Protocol

getAddress()

Returns a string containing `Agent`'s address.

Syntax

```
string getAddress() const;
```

getName()

Returns a string containing `Agent`'s name.

Syntax

```
string getName() const;
```

getProtocol()

Returns a numeric code representing `Agent`'s protocol.

Syntax

```
unsigned int getProtocol();
```

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();
```

setAddress()

Set the address of the `Agent` object.

Syntax

```
void setAddress(
   const string& address);
```

Parameter	Description
`address`	The name of the `Agent` object.

setName()

Set the name of the `Agent` object.

Syntax

```
void setName(
   const string& name);
```

Parameter	Description
`name`	The name of the `Agent` object.

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();
```

setProtocol()

Set the protocol of the `Agent` object.

Syntax

```
void setProtocol(
   unsigned int protocol = 0);
```

Parameter	Description
`protocol`	The protocol of the `Agent` object.

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.

Note: See Table 10–3, " OCCI Datatypes supported by AnyData Class" for supported datatypes.

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 10–1 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 10–2 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 prior to conversion.

```
Connection *conn;
string str;
...
```

```
if(!any.isNULL())
{   if(any.getType()==OCCI_TYPECODE_VARCHAR2)
   {
      str = any.getAsString();
      cout<<str;
   }
}
...
```

Example 10–3   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 10–4   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 prior to 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 10–3 OCCI Datatypes supported by AnyData Class

Datatype	TypeCode
`BDouble`	`OCCI_TYPECODE_BDOUBLE`
`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 10–4 Summary of AnyData Methods

Method	Summary
AnyData() on page 10-18	`AnyData` class constructor.
getAsBDouble() on page 10-18	Converts an `AnyData` object into `BDouble`.
getAsBfile() on page 10-18	Converts an `AnyData` object into `Bfile`.
getAsBFloat() on page 10-18	Converts an `AnyData` object into `BFloat`.
getAsBytes() on page 10-19	Converts an `AnyData` object into `Bytes`.
getAsDate() on page 10-19	Converts an `AnyData` object into `Date`.

Table 10–4 (Cont.) Summary of AnyData Methods

Method	Summary
getAsIntervalDS() on page 10-19	Converts an AnyData object into IntervalDS.
getAsIntervalYM() on page 10-19	Converts an AnyData object into IntervalYM.
getAsNumber() on page 10-19	Converts an AnyData object into Number.
getAsObject() on page 10-20	Converts an AnyData object into PObject.
getAsRef() on page 10-20	Converts an AnyData object into RefAny.
getAsString() on page 10-20	Converts an AnyData object into a namespace string.
getAsTimestamp() on page 10-20	Converts an AnyData object into Timestamp.
getType() on page 10-20	Retrieves the DataType held by the AnyData object. See Table 10–3.
isNull() on page 10-21	Tests whether AnyData object is NULL.
setFromBDouble() on page 10-21	Converts a BDouble into Anydata.
setFromBfile() on page 10-21	Converts a Bfile into Anydata.
setFromBFloat() on page 10-21	Converts a BFloat into Anydata.
setFromBytes() on page 10-22	Converts a Bytes into Anydata.
setFromDate() on page 10-22	Converts a Date into Anydata.
setFromIntervalDS() on page 10-22	Converts an IntervalDS into Anydata.
setFromIntervalYM() on page 10-23	Converts an IntervalYM into Anydata.
setFromNumber() on page 10-23	Converts a Number into Anydata.
setFromObject() on page 10-23	Converts a PObject into Anydata.
setFromRef() on page 10-24	Converts a RefAny into Anydata.
setFromString() on page 10-24	Converts a namespace string into Anydata.
setFromTimestamp() on page 10-24	Converts a Timestamp into Anydata.
setNull() on page 10-25	Sets AnyData object to NULL.

AnyData()

AnyData constructor.

Syntax

```
AnyData(
   const Connection *sessp);
```

Parameter	Description
sessp	The connection.

getAsBDouble()

Converts an AnyData object into BDouble.

Syntax

```
BDouble getAs2BDouble() const;
```

getAsBfile()

Converts an AnyData object into Bfile.

Syntax

```
Bfile getAsBfile() const;
```

getAsBFloat()

Converts an AnyData object into BFloat.

Syntax

```
BFloat getAsBFloat() const;
```

getAsBytes()

Converts an `AnyData` object into Bytes.

Syntax

```
Bytes getAsBytes() const;
```

getAsDate()

Converts an `AnyData` object into `Date`.

Syntax

```
Date getAsDate() const;
```

getAsIntervalDS()

Converts an `AnyData` object into `IntervalDS`.

Syntax

```
IntervalDS getAsIntervalDS() const;
```

getAsIntervalYM()

Converts an `AnyData` object into `IntervalYM`.

Syntax

```
IntervalYS getAsIntervalYM() const;
```

getAsNumber()

Converts an `AnyData` object into `Number`.

Syntax

```
Number getAsNumber() const;
```

getAsObject()

Converts an `AnyData` object into `PObject`.

Syntax

```
PObject* getAsObject() const;
```

getAsRef()

Converts an `AnyData` object into `RefAny`.

Syntax

```
RefAny getAsRef() const;
```

getAsString()

Converts an `AnyData` object into a namespace `string`.

Syntax

```
string getAsString() const;
```

getAsTimestamp()

Converts an `AnyData` object into `Timestamp`.

Syntax

```
Timestamp getAsTimestamp() const;
```

getType()

Retrieves the data type held by the `AnyData` object. Refer to Table 10–3 on page 10-16 for valid values for `TypeCode`.

Syntax

```
TypeCode getType();
```

isNull()

Tests whether the `AnyData` object is `NULL`. If the `AnyData` object is `NULL`, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

setFromBDouble()

Converts a `BDouble` into `AnyData`.

Syntax

```
void setFromBDouble(
   const BDouble& bd);
```

Parameter	Description
`bd`	The `BDouble` that will be converted into `AnyData`.

setFromBfile()

Converts a `Bfile` into `AnyData`.

Syntax

```
void setFromBfile(
   const Bfile& bfile);
```

Parameter	Description
`bfile`	The `Bfile` that will be converted into `AnyData`.

setFromBFloat()

Converts a `BFloat` into `AnyData`.

Syntax

```
void setFromBFloat(
   const BFloat& bf);
```

Parameter	Description
bf	The `BFloat` that will be converted into `AnyData`.

setFromBytes()

Converts a `Bytes` into `AnyData`.

Syntax

```
void setFromBytes(
   const Bytes& bytes);
```

Parameter	Description
bytes	The `Bytes` that will be converted into `AnyData`.

setFromDate()

Converts a `Date` into `AnyData`.

Syntax

```
void setFromDate(
   const Date& date);
```

Parameter	Description
date	The `Date` that will be converted into `AnyData`.

setFromIntervalDS()

Converts an `IntervalDS` into `AnyData`.

Syntax

```
void setFromIntervalDS(
   const IntervalDS& invDS);
```

Parameter	Description
`invDS`	The `IntervalDS` that will be converted into `AnyData`.

setFromIntervalYM()

Converts an `IntervalYM` into `AnyData`.

Syntax

```
void setFromIntervalYM(
   const IntervalYM& invYM);
```

Parameter	Description
`invYM`	The `IntervalYM` that will be converted into `AnyData`.

setFromNumber()

Converts a `Number` into `AnyData`.

Syntax

```
void setFromNumber(
   const Number& num);
```

Parameter	Description
`num`	The `Number` that will be converted into `AnyData`.

setFromObject()

Converts a `PObject` into `AnyData`.

Syntax

```
void setFromObject(
   const PObject* objptr);
```

Parameter	Description
`objptr`	The `PObject` that will be converted into `AnyData`.

setFromRef()

Converts a `PObject` into `AnyData`.

Syntax

```
void setFromRef(
   const RefAny& ref);
```

Parameter	Description
`ref`	The `RefAny` that will be converted into `AnyData`.

setFromString()

Converts a namespace `string` into `AnyData`.

Syntax

```
void setFromString(
   string& str);
```

Parameter	Description
`str`	The namespace `string` that will be converted into `AnyData`.

setFromTimestamp()

Converts a `Timestamp` into `AnyData`.

Syntax

```
void setFromTimestamp(
   const Timestamp& timestamp);
```

Parameter	Description
`timestamp`	The `Timestamp` that will be converted into `AnyData`.

setNull()

Sets `AnyData object` to `NULL`.

Syntax

```
void setNull();
```

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 the Oracle database. A `Bfile` object contains a logical pointer to a `BFILE`, not the `BFILE` itself.

Methods of the `Bfile` class enable you to perform specific tasks related to `Bfile` objects.

Methods of the `ResultSet` and `Statement` classes, such as `getBfile()` and `setBfile()`, enable you to access an SQL `BFILE` value.

The only methods valid on a `NULL` `Bfile` object are setName(), isNull(), and operator=().

An uninitialized `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 will modify the `BFILE`; however, the modified data will be flushed to the table only when the transaction is committed. Note that an insert is not required.
- Assigning an initialized `Bfile` object to it.

See Also:

- In-depth discussion of LOBs in the introductory chapter of *Oracle Database Application Developer's Guide - Large Objects*,

Table 10–5 Summary of Bfile Methods

Method	Summary
Bfile() on page 10-27	`Bfile` class constructor.
close() on page 10-28	Close a previously opened `BFILE`.
closeStream() on page 10-28	Close the stream obtained from the `BFILE`.
fileExists() on page 10-28	Test whether the `BFILE` exists.
getDirAlias() on page 10-28	Return the directory object of the `BFILE`.
getFileName() on page 10-29	Return the name of the `BFILE`.
getStream() on page 10-29	Return data from the `BFILE` as a `Stream` object.

Table 10–5 (Cont.) Summary of Bfile Methods

Method	Summary
getUStringDirAlias() on page 10-29	Return a `UString` containing the directory object associated with the `BFILE`.
getUStringFileName() on page 10-30	Return a `UString` containing the file name associated with the `BFILE`.
isInitialized() on page 10-30	Test whether the `Bfile` object is initialized.
isNull() on page 10-30	Test whether the `Bfile` object is atomically `NULL`.
isOpen() on page 10-30	Test whether the `BFILE` is open.
length() on page 10-31	Return the number of bytes in the `BFILE`.
open() on page 10-31	Open the `BFILE` with read-only access.
operator=() on page 10-31	Assign a `BFILE` locator to the `Bfile` object.
operator==() on page 10-32	Test whether two `Bfile` objects are equal.
operator!=() on page 10-32	Test whether two `Bfile` objects are not equal.
operator==() on page 10-32	Read a specified portion of the `BFILE` into a buffer.
setName() on page 10-33	Set the directory object and file name of the `BFILE`.
setNull() on page 10-33	Set the `Bfile` object to atomically `NULL`.

Bfile()

`Bfile` class constructor.

Syntax	Description
`Bfile();`	Creates a `NULL` `Bfile` object.

`Bfile(`
`   const Connection *connectionp);`
	Create an uninitialized `Bfile` object.

`Bfile(`
`   const Bfile &srcBfile);`
	Create a copy of a `Bfile` object.

Parameter	Description
`connectionp`	The connection pointer

Parameter	Description
`srcBfile`	The source `Bfile` object

close()

Closes a previously opened `Bfile`.

Syntax

```
void close();
```

closeStream()

Closes the stream obtained from the `Bfile`.

Syntax

```
void closeStream(
   Stream *stream);
```

Parameter	Description
`stream`	The stream to ne closed.

fileExists()

Tests whether the `BFILE` exists. If the `BFILE` exists, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool fileExists() const;
```

getDirAlias()

Returns a string containing the directory object associated with the `BFILE`.

Syntax

```
string getDirAlias() const;
```

getFileName()

Returns a string containing the file name associated with the `BFILE`.

Syntax

```
string getFileName() const;
```

getStream()

Returns a `Stream` object read from the `BFILE`. If a stream is already open, it is disallowed to open another stream on the `Bfile` object. The stream must be closed before performing any `Bfile` 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 `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 will be read in a streamed mode from input `offset` until the end of the `BFILE`.

getUStringDirAlias()

Return 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 charset.

Syntax

```
UString getUStringDirAlias() const;
```

getUStringFileName()

Return a `UString` containing the file name associated with the `BFILE`.

> **Note:** The `UString` object is in UTF16 charset. The environment associated with `BFILE` should be associated with UTF16 charset.

Syntax

```
UString getUStringFileName() const;
```

isInitialized()

Tests whether the `Bfile` object has been initialized. If the `Bfile` object has been initialized, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isInitialized() const;
```

isNull()

Tests whether the `Bfile` object is atomically `NULL`. If the `Bfile` object is atomically `NULL`, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

isOpen()

Tests whether the `BFILE` is open. The `BFILE` is considered to be open only if it was opened by a call on this `Bfile` object. (A different `Bfile` object could have opened this file as more than one open can be performed on the same file by associating the file with different `Bfile` objects). If the `BFILE` is open, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isOpen() const;
```

length()

Returns the number of bytes (inclusive of the end of file marker) in the `BFILE`.

Syntax

```
unsigned int length() const;
```

open()

Opens an existing `BFILE` for read-only access. This function is meaningful the first time it is called for a `Bfile` object.

Syntax

```
void open();
```

operator=()

Assigns a `Bfile` object to the current `Bfile` object. The source `Bfile` object is assigned to this `Bfile` object only when this `Bfile` object gets stored in the database.

Syntax

```
Bfile& operator=(
   const Bfile &srcBfile);
```

Parameter	**Description**
`srcBfile`	The `Bfile` object to be assigned to the current `Bfile` object.

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;
```

Parameter	Description
`srcBfile`	The `Bfile` object to be compared with the current `Bfile` object.

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.

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.

setName()

Sets the directory object and file name of the BFILE.

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.

setNull()

Sets the Bfile object to atomically NULL.

Syntax

```
void setNull();
```

Blob Class

The Blob class defines the common properties of objects of type BLOB. A BLOB is a large binary object stored as a column value in a row of a database table. A Blob object contains a logical pointer to a BLOB, not the BLOB itself.

Methods of the Blob class enable you to perform specific tasks related to Blob objects.

Methods of the ResultSet and Statement classes, such as getBlob() and setBlob(), enable you to access an SQL BLOB value.

The only methods valid on a NULL Blob object are setName(), isNull(), and operator=().

An uninitialized Blob object can be initialized by:

- The setEmpty() method. The BLOB can then be modified by inserting this BLOB into the table and then retrieving it using SELECT ... FOR UPDATE. The write() method will modify the BLOB; however, the modified data will be 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 the introductory chapter of *Oracle Database Application Developer's Guide - Large Objects*,

Table 10–6 Summary of Blob Methods

Method	Summary
Blob() on page 10-36	Blob class constructor.
append() on page 10-37	Append a specified BLOB to the end of the current BLOB.
close() on page 10-28	Close a previously opened BLOB.
closeStream() on page 10-28	Close the Stream object obtained from the BLOB.
copy() on page 10-38	Copy a specified portion of a BFILE or BLOB into the current BLOB.
getChunkSize() on page 10-38	Return the chunk size of the BLOB.

Table 10–6 (Cont.) Summary of Blob Methods

Method	**Summary**
getStream() on page 10-29	Return data from the BLOB as a `Stream` object.
isInitialized() on page 10-30	Test whether the `Blob` object is initialized
isNull() on page 10-30	Test whether the `Blob` object is atomically `NULL`.
isOpen() on page 10-30	Test whether the BLOB is open
length() on page 10-31	Return the number of bytes in the BLOB.
open() on page 10-31	Open the BLOB `with read or` read/write access.
operator=() on page 10-31	Assign a BLOB locator to the `Blob` object.
operator==() on page 10-41	Test whether two `Blob` objects are equal.
operator!=() on page 10-32	Test whether two `Blob` objects are not equal.
operator==() on page 10-32	Read a portion of the BLOB into a buffer.
setEmpty() on page 10-42	Set the `Blob` object to empty.
setEmpty() on page 10-42	Set the `Blob` object to empty and initializes the connection pointer to the passed parameter.
setName() on page 10-33	Set the `Blob` object to atomically `NULL`.
trim() on page 10-43	Truncate the BLOB to a specified length.
write() on page 10-43	Write a buffer into an *unopened* BLOB.
writeChunk() on page 10-44	Write a buffer into an *open* BLOB.

Blob()

`Blob` class constructor.

Syntax	**Description**
`Blob();`	Creates a `NULL Blob` object.

`Blob(`
`   const Connection *connectionp);`
	Create an uninitialized `Blob` object.

`Blob(`
`   const Blob &srcBlob);`
	Create a copy of a `Blob` object.

Parameter	Description
connectionp	The connection pointer
srcBlob	The source Blob object.

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.

close()

Closes a BLOB.

Syntax

```
void close();
```

closeStream()

Closes the Stream object obtained from the BLOB.

Syntax

```
void closeStream(
   Stream *stream);
```

Parameter	Description
stream	The Stream to be closed.

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.

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.

getChunkSize()

Returns the chunk size of the BLOB. When creating a table that contains a BLOB, the user can specify the chunking factor, which can be a multiple of Oracle blocks. This corresponds to the chunk size used by the LOB data layer when accessing or modifying the BLOB.

Syntax

```
unsigned int getChunkSize() const;
```

getStream()

Returns a Stream object from the BLOB. If a stream is already 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 will be read in a streamed mode from input offset until the end of the BLOB.

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;
```

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;
```

isOpen()

Tests whether the BLOB is open. If the BLOB is open, then TRUE is returned; otherwise, FALSE is returned.

Syntax

```
bool isOpen() const;
```

length()

Returns the number of bytes in the BLOB.

Syntax

```
unsigned int length() const;
```

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

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.

operator==()

Compares two Blob objects for equality. Two Blob objects are equal if they both refer to the same BLOB. Two NULL Blob objects are not considered equal. If the Blob objects are equal, then TRUE is returned; otherwise, FALSE is returned.

Syntax

```
bool operator==(
   const Blob &srcBlob) const;
```

Parameter	Description
srcBlob	The source BLOB to be compared with the current BLOB.

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.

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`.

setEmpty()

Sets the `Blob` object to empty.

Syntax

```
void setEmpty();
```

setEmpty()

Sets the `Blob` object to empty and initializes the connection pointer to the passed parameter.

Syntax

```
void setEmpty(
   const Connection* connectionp);
```

Parameter	Description
connectionp	The new connection pointer for the Blob object.

setNull()

Sets the Blob object to atomically NULL.

Syntax

```
void setNull();
```

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.

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 already 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.

writeChunk()

Writes data from a buffer into a previously opened BLOB. The actual number of bytes written is returned.

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.

Bytes Class

Methods of the Bytes class enable you to perform specific tasks related to Bytes objects.

Table 10–7 Summary of Bytes Methods

Method	Summary
Bytes() on page 10-45	Bytes class constructor.
byteAt() on page 10-46	Return the byte at the specified position of the Bytes object.
getBytes() on page 10-46	Return a byte array from the Bytes object.
isNull() on page 10-31	Test whether the Bytes object is NULL.
length() on page 10-31	Return the number of bytes in the Bytes object.
setNull() on page 10-47	Set the Bytes object to NULL.

Bytes()

Bytes class constructor.

Syntax	Description
`Bytes( Environment *env = NULL);`	Create a Bytes object.
`Bytes( unsigned char *value, unsigned int count; unsigned int offset = 0, Environment *env = NULL);`	Creates a Bytes object that contains a subarray of bytes from a character array.
`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

Parameter	Description
`count`	The size of the subset of the character array that will be copied into the new bytes object
`offset`	The first position from which to begin copying the character array
`e`	The source `Bytes` object.

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.

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.
`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.

Parameter	Description
`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.

isNull()

Tests whether the `Bytes` object is atomically `NULL`. If the `Bytes` object is atomically `NULL`, then `TRUE` is returned; otherwise `FALSE` is returned.

Syntax

```
bool isNull() const;
```

length()

This method returns the length of the `Bytes` object.

Syntax

```
unsigned int length() const;
```

setNull()

This method sets the `Bytes` object to atomically `NULL`.

Syntax

```
void setNull();
```

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.

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 will modify the `CLOB`; however, the modified data will be flushed to the table only when the transaction is committed. Note that an `insert` is not required.
- Assigning an initialized `Clob` object to it.

See Also:

- In-depth discussion of LOBs in the introductory chapter of *Oracle Database Application Developer's Guide - Large Objects*,

Table 10–8 Summary of Clob Methods

Method	Summary
Clob() on page 10-49	`Clob` class constructor.
append() on page 10-50	Append a `Clob` at the end of the current `Clob`.
close() on page 10-50	Close a previously opened `Clob`.
closeStream() `on page 10-50`	Close the `Stream` object obtained from the current `Clob`.
copy() on page 10-51	Copy all or a portion of a `Clob` or `BFILE` into the current `Clob`.

Table 10–8 (Cont.) Summary of Clob Methods

Method	Summary
getCharSetForm() on page 10-52	Return the character set form of the `Clob`.
getCharSetId() on page 10-52	Return the character set ID of the `Clob`.
getChunkSize() on page 10-52	Return the chunk size of the `Clob`.
getStream() on page 10-52	Return data from the `CLOB` as a `Stream` object.
isInitialized() on page 10-53	Test whether the `Clob` object is initialized.
isNull() on page 10-53	Test whether the `Clob` object is atomically `NULL`.
isOpen() on page 10-53	Test whether the `Clob` is open.
length() on page 10-53	Return the number of characters in the current `CLOB`.
open() on page 10-54	Open the `CLOB` with read or read/write access.
operator=() on page 10-54	Assign a `CLOB` locator to the current `Clob` object.
operator==() on page 10-54	Test whether two `Clob` objects are equal.
operator!=() on page 10-55	Test whether two `Clob` objects are not equal.
read() on page 10-55	Read a portion of the `CLOB` into a buffer.
setCharSetId() on page 10-56	Sets the character set ID associated with the `Clob`.
setCharSetIdUString() on page 10-57	Sets the character set ID associated with the `Clob`; used when the environment character set is UTF16.
setCharSetForm() on page 10-57	Sets the character set form associated with the `Clob`.
setEmpty() on page 10-57	Set the `Clob` object to empty.
setEmpty() on page 10-58	Set the `Clob` object to empty and initialize the connection pointer to the passed parameter.
setNull() on page 10-58	Set the `Clob` object to atomically `NULL`.
trim() on page 10-58	Truncate the `Clob` to a specified length.
write() on page 10-59	Write a buffer into an *unopened* `CLOB`.
writeChunk() on page 10-59	Write a buffer into an *open* `CLOB`.

Clob()

`Clob` class constructor.

Syntax	Description
`Clob();`	Creates a `NULL` `Clob` object.

`Clob(`
`   const Connection *connectionp);`
	Create an uninitialized `Clob` object.

`Clob(`
`   const Clob *srcClob);`
	Create a copy of a `Clob` object.

Parameter	Description
`connectionp`	Connection pointer
`srcClob`	The source `Clob` object

append()

Appends a `CLOB` to the end of the current `CLOB`.

Syntax

```
void append(
   const Clob &srcClob);
```

Parameter	Description
`srcClob`	The `CLOB` to be appended to the current `CLOB`.

close()

Closes a `CLOB`.

Syntax

```
void close();
```

closeStream()

Closes the `Stream` object obtained from the `CLOB`.

Syntax

```
void closeStream(
   Stream *stream);
```

Parameter	Description
stream	The Stream object to be closed.

copy()

Copies a part or all of a BFILE or CLOB into the current CLOB.

Syntax	Description

`void copy(`
`   const Bfile &srcBfile,`
`   unsigned int numBytes,`
`   unsigned int dstOffset = 1,`
`   unsigned int srcOffset = 1);`
	Copies a BFILE into the current CLOB.

`void copy(`
`   const Blob &srcClob,`
`   unsigned int numBytes,`
`   unsigned int dstOffset = 1,`
`   unsigned int srcOffset = 1);`
	Copies a CLOB into the current 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 characters 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.

getCharSetForm()

Returns the character set form of the CLOB.

Syntax

```
CharSetForm getCharSetForm() const;
```

getCharSetId()

Returns the character set ID of the CLOB, in string form.

Syntax

```
string getCharSetId() const;
```

getChunkSize()

Returns the chunk size of the CLOB. When creating a table that contains a CLOB, the user can specify the chunking factor, which can be a multiple of Oracle blocks. This corresponds to the chunk size used by the LOB data layer when accessing and modifying the CLOB.

Syntax

```
unsigned int getChunkSize() const;
```

getStream()

Returns a Stream object from the CLOB. If a stream is already 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 will be 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 will be read from the `offset` value until the end of the `CLOB`.

isInitialized()

Tests whether the `Clob object` is initialized. If the `Clob` object is initialized, `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isInitialized() const;
```

isNull()

Tests whether the `Clob` object is atomically `NULL`. If the `Clob` object is atomically `NULL`, `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

isOpen()

Tests whether the `CLOB` is open. If the `CLOB` is open, `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isOpen() const;
```

length()

Returns the number of characters in the `CLOB`.

Syntax

```
unsigned int length() const;
```

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

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);
```

Parameter	Description
srcClob	The Clob from which the data must be copied.

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.

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.

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 will be used by default, unless they are explicitly set through setCharSetId(), setCharSetIdUString() and setCharSetForm() calls.

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.

Syntax	Description

`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.

Note: 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).

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.

setCharSetId()

Sets the Character set Id associated with `Clob`. The charset id set will be 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 (WE8DEC, ZHT16BIG5), or "OCCIUTF16".

setCharSetIdUString()

Sets the Character set Id associated with `Clob`; used when the environment's charset is UTF16. The charset id set will be used for read/write and getStream() operations.

Syntax

```
void setCharSetIdUSString(
   const string &charset);
```

Parameter	Description
`charset`	Oracle supported characterset name (WE8DEC, ZHT16BIG5), or "OCCIUTF16" in `UString` (UTF16 characterset).

setCharSetForm()

Sets the character set form associated with the `CLOB`. The charset form set will be used for read/write and getStream() operations. If no value is set explicitly, by default, `OCCI_SQLCS_IMPLICIT` will be used.

Syntax

```
void setCharSetForm(
   CharSetForm csfrm );
```

Parameter	Description
`csfrm`	The char set form for `Clob`.

setEmpty()

Sets the `Clob` object to empty.

Syntax

```
void setEmpty();
```

setEmpty()

Sets the `Clob` object to empty and initializes the connection pointer to the passed parameter.

Syntax

```
void setEmpty(
   const Connection* connectionp);
```

Parameter	Description
`connectionp`	The new connection pointer for the `Clob` object.

setNull()

Sets the `Clob` object to atomically `NULL`.

Syntax

```
void setNull();
```

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`.

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 already open, use writeChunk() instead. The actual number of characters written is returned. The client's character set id and form will be used by default, unless they are explicitly set through setCharSetId() and setCharSetForm() calls.

Syntax	Description

`unsigned int write(`
`   unsigned int amt,`
`   unsigned char *buffer,`
`   unsigned int bufsize,`
`   unsigned int offset=1) const;`
	Writes data from a buffer into a CLOB.

`unsigned int write(`
`   unsigned int amt,`
`   utext *buffer,`
`   unsigned int bufsize,`
`   unsigned int offset=1) const;`
	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.

writeChunk()

Writes data from a buffer into a previously opened CLOB.

The actual number of characters written is returned.   The client's character set id and form will be 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) const;`
	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) const;`
	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.

Connection Class

The Connection class represents a connection with a specific database. Within the context of a connection, SQL statements are executed and results are returned.

Table 10–9 Summary of Connection Methods

Method	Summary
changePassword() on page 10-62	Change the password for the current user.
commit() on page 10-63	Commit changes made since the previous commit or rollback and release any database locks held by the session.
createStatement() on page 10-63	Create a `Statement` object to execute SQL statements.
flushCache() on page 10-64	Flush the object cache associated with the connection.
getClientCharSet() on page 10-64	Return the default client character set.
getClientCharSetUString() on page 10-64	Return the globalization enabled client character set in `UString`.
getClientNCHARCharSet() on page 10-64	Return the default client NCHAR character set.
getClientNCHARCharSetUString() on page 10-65	Return the globalization enabled client NCHAR character set in `UString`.
getMetaData() on page 10-65	Return the metadata for an object accessible from the connection.
getOCIServer() on page 10-66	Return the OCI server context associated with the connection.
getOCIServiceContext() on page 10-67	Return the OCI service context associated with the connection.
getOCISession() on page 10-67	Return the OCI session context associated with the connection.
getStmtCacheSize() on page 10-67	Retrieves the size of the statement cache.
getTag() on page 10-67	Returns the tag associated with the connection.
isCached() on page 10-67	Determines if the specified statement is cached.
pinVectorOfRefs() on page 10-68	Pins an entire vector of `Ref` objects into object cache in a single round trip; pinned objects are available through an `OUT` parameter vector.

Table 10–9 (Cont.) Summary of Connection Methods

Method	Summary
registerSubscriptions() on page 10-70	Register several `Subscriptions` for notification.
rollback() on page 10-69	Roll back all changes made since the previous commit or rollback and release any database locks held by the session.
postToSubscriptions() on page 10-69	Posts notifications to subscriptions.
setStmtCacheSize() on page 10-70	Enables or disables statement caching.
terminateStatement() on page 10-71	Close a `Statement` object and free all resources associated with it.
unregisterSubscription() on page 10-71	Unregisters a `Subscription`, turning off its notifications

changePassword()

Changes the password of the user currently connected to the database.

Syntax	Description

`void changePassword(`
`   const string &user,`
`   const string &oldPassword,`
`   const string &newPassword);`
	Changes the password of the user.

`void changePassword(`
`   const UString &user,`
`   const UString &oldPassword,`
`   const UString &newPassword);`
	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.

commit()

Commits all changes made since the previous commit or rollback, and releases any database locks currently held by the session.

Syntax

```
void commit();
```

createStatement()

Creates a `Statement` object with the SQL statement specified.

Syntax	Description
`Statement* createStatement( const string &sql);`	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);`	Searches the cache for a statement with a matching tag; if not found, creates a new statement with the specified SQL content.
`Statement* createStatement( const UString &sql);`	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);`	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 needs to be retrieved from the cache. Ignored if statement caching is disabled.

Note:

- 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`.
- Non-caching versions of this method always create and return a new statement

---

flushCache()

Flushes the object cache associated with the connection.

Syntax

```
void flushCache();
```

getClientCharSet()

Returns the session's character set.

Syntax

```
string getClientCharSet() const;
```

getClientCharSetUString()

Returns the globalization enabled client character set in `UString`.

Syntax

```
UString getClientCharSetUString() const;
```

getClientNCHARCharSet()

Returns the session's `NCHAR` character set.

Syntax

```
string getClientNCHARCharSet() const;
```

getClientNCHARCharSetUString()

Returns the globalization enabled client `NCHAR` character set in `UString`.

Syntax

```
UString getClientNCHARCharSetUString() const;
```

getMetaData()

Returns metadata for an object in the database.

Syntax	Description

`MetaData getMetaData(`
`   const string &object,`
`   MetaData::ParamType prmtyp=MetaData::PTYPE_UNK) const;`
	Returns metadata for an object in the database.

`MetaData getMetaData(`
`   const UString &object,`
`   MetaData::ParamType prmtyp=MetaData::PTYPE_UNK) const;`
	Returns metadata for a globalization enabled object in the database.

`MetaData getMetaData(`
`   const RefAny &ref);`
	Returns metadata for an object in the database through a reference.

Parameter	Description
obj	The SQL string to be associated with the statement object.

Parameter	Description
prmtyp	The type of the schema object being described. The possible values for this are enumerated by `MetaData::ParamType`. Valid 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`—the results of a method ■ `PTYPE_SCHEMA`—schema ■ `PTYPE_DATABASE`—database ■ `PTYPE_UNK`—type unknown
ref	A `REF` to the Type Descriptor Object (TDO) of the type to be described.

getOCIServer()

Returns the OCI server context associated with the connection.

Syntax

```
LNOCIServer* getOCIServer() const;
```

getOCIServiceContext()

Returns the OCI service context associated with the connection.

Syntax

```
LNOCISvcCtx* getOCIServiceContext() const;
```

getOCISession()

Returns the OCI session context associated with the connection.

Syntax

```
LNOCISession* getOCISession() const;
```

getStmtCacheSize()

Retrieves the size of the statement cache.

Syntax

```
unsigned int getStmtCacheSize() const;
```

getTag()

Returns the tag associated with the connection. Valid only for connections from a stateless connection pool.

Syntax

```
string getTag() const;
```

isCached()

Determines if the specified statement is cached.

Syntax	Description

`bool isCached(`
`   const string &sql,`
`   const string &tag="");`
	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="");`
	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 needs to be retrieved from the cache. Ignored if statement caching is disabled.

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.

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 needs to dereference a particular `Ref` object by a `ptr()` call, which returns a previously pinned object.

Syntax

```
template <class T> void pinVectorOfRefs(
   const Connection *conn,
   vector <Ref<T>> & vect,
   vector <T*> &vectObj,
   LockOptions lockOpt);
```

Parameter	Description
`conn`	Connection
`vect`	Vector of `Ref` objects that will be pinned.
`vectObj`	Vector that will contain objects after the pinning operation is complete; an `OUT` parameter.
`lockOpt`	Lock option used during the pinning of the array. The only value supported at this time is `OCCI_LOCK_NONE`.

rollback()

Drops all changes made since the previous commit or rollback, and releases any database locks currently held by the session.

Syntax

```
void rollback();
```

postToSubscriptions()

Posts notifications to subscriptions.

The `Subscription` object needs to have a valid subscription name, and the namespace should be set to `NS_ANONYMOUS`. The payload needs to 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 in the case of several system events. If the application needs more rigid guarantees, it can use the Streams AQ functionality.

Syntax

```
void postToSubscriptions(
   const vector<Subscription>& sub);
```

Parameter	Description
`sub`	The vector of subscriptions that receive postings.

registerSubscriptions()

Registers `Subscriptions` for notification.

New client processes and existing processes that restart after a shut down must register for all subscriptions of interest. If the client stays up during a server shut down and restart, this client will continue 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<Subscription>& sub_list);
```

Parameter	Description
`sub_list`	Vector of subscriptions that will be registered for notification.

setStmtCacheSize()

Enables or disables statement caching.

- A nonzero value will enable statement caching, with a cache of specified size.
- A zero value will disable caching.

Syntax

```
void setStmtCacheSize(
   unsigned int size);
```

Parameter	Description
`size`	The maximum number of statements in the cache.

terminateStatement()

Closes a `Statement` object.

Syntax	Description
`void terminateStatement( Statement *statement);`	Closes a `Statement` object and frees all resources associated with it.
`void terminateStatement( Statement *statement, const string tag);`	Release statement back to the cache after tagging it with the optional tag.

Parameter	Description
`statement`	The `Statement` to be closed.
`tag`	The tag associated with the statement.

unregisterSubscription()

Unregisters a `Subscription`, turning off its notifications.

Syntax

```
void unregisterSubscription(
   const Subscription& sub);
```

Parameter	Description
`sub`	`Subscription` whose notifications will be turned off.

ConnectionPool Class

The ConnectionPool class represents a pool of connections for a specific database.

Table 10–10 Summary of ConnectionPool Methods

Method	Summary
createConnection() on page 10-73	Create a pooled connection.
createProxyConnection() on page 10-73	Create a proxy connection.
getBusyConnections() on page 10-74	Return the number of busy connections in the connection pool.
getIncrConnections() on page 10-74	Return the number of incremental connections in the connection pool.
getMaxConnections() on page 10-74	Return the maximum number of connections in the connection pool.
getMinConnections() on page 10-75	Return the minimum number of connections in the connection pool.
getOpenConnections() on page 10-75	Return the number of open connections in the connection pool.
getPoolName() on page 10-75	Return the name of the connection pool.
getStmtCacheSize() on page 10-75	Retrieves the size of the statement cache.
getTimeOut() on page 10-75	Return the time out period for a connection in the connection pool.
setErrorOnBusy() on page 10-76	Specify that a `SQLException` should be generated when all connections in the connection pool are busy and no further connections can be opened.
setPoolSize() on page 10-76	Set the minimum, maximum, and incremental number of pooled connections for the connection pool.
setStmtCacheSize() on page 10-75	Retrieves the size of the statement cache.
setTimeOut() on page 10-76	Set the time out period, in seconds, for a connection in the connection pool.
terminateConnection() on page 10-77	Destroy the connection.

createConnection()

Creates a pooled connection.

Syntax	Description

`Connection* createConnection(`
`   const string &userName,`
`   const string &password);`
	Creates a pooled connection.

`Connection* createConnection(`
`   const UString &username,`
`   const UString &password);`
	Creates a globalization enabled pooled connection.

Parameter	Description
`userName`	The name of the user with which to connect.
`password`	The password of the user.

createProxyConnection()

Creates a proxy connection from the connection pool.

Syntax	Description

`Connection* createProxyConnection(`
`   const string &name,`
`   Connection::ProxyType proxyType=Connection::PROXY_DEFAULT);`
	Creates a proxy connection.

`Connection* createProxyConnection(`
`   const UString &name,`
`   Connection::ProxyType proxyType=Connection::PROXY_DEFAULT);`
	Creates a globalization enabled proxy connection.

`Connection* createProxyConnection(`
`   const string &name,`
`   string roles[],`
`   int numRoles,`
`   Connection::ProxyType proxyType=Connection::PROXY_DEFAULT);`
	Creates a proxy connection for several roles.

Syntax	Description

`Connection* createProxyConnection(`
`   const UString &name,`
`   std::string roles[],`
`   unsigned int numRoles,`
`   Connection::ProxyType proxyType=Connection::PROXY_DEFAULT);`
	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.
`numolesR`	The number of roles to activate on the database server.
`numolesR`	The type of proxy authentication to perform. Valid values are: ■ `PROXY_DEFAULT` representing a database user name. ■ `PROXY_EXTERNAL_AUTH` representing an external user name.

getBusyConnections()

Returns the number of busy connections in the connection pool.

Syntax

```
unsigned int getBusyConnections() const;
```

getIncrConnections()

Returns the number of incremental connections in the connection pool.

Syntax

```
unsigned int getIncrConnections() const;
```

getMaxConnections()

Returns the maximum number of connections in the connection pool.

Syntax

```
unsigned int getMaxConnections() const;
```

getMinConnections()

Returns the minimum number of connections in the connection pool.

Syntax

```
unsigned int getMinConnections() const;
```

getOpenConnections()

Returns the number of open connections in the connection pool.

Syntax

```
unsigned int getOpenConnections() const;
```

getPoolName()

Returns the name of the connection pool.

Syntax

```
string getPoolName() const;
```

getStmtCacheSize()

Retrieves the size of the statement cache.

Syntax

```
unsigned int getStmtCacheSize() const;
```

getTimeOut()

Returns the time out period of a connection in the connection pool.

Syntax

```
unsigned int getTimeOut() const;
```

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();
```

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);
```

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.

setStmtCacheSize()

Enables or disables statement caching.

- A nonzero value will enable statement caching, with a cache of specified size.
- A zero value will disable caching.

Syntax

```
void setStmtCacheSize(
   unsigned int size);
```

Parameter	Description
`size`	The size of the statement cache.

setTimeOut()

Sets the time out period for a connection in the connection pool. OCCI will terminate any connections related to this connection pool that have been idle for longer than the time out period specified.

Syntax

```
void setTimeOut(
   unsigned int connTimeOut = 0);
```

Parameter	Description
`connTimeOut`	The timeout period in number of seconds.

terminateConnection()

Terminates the pooled connection or proxy connection.

Syntax

```
void terminateConnection(
   Connection *connection);
```

Parameter	Description
`connection`	The pooled connection or proxy connection to terminate.

Consumer Class

The Consumer class supports dequeuing of Messages and controls the dequeuing options.

For typical usage, please see Example 10–5, Example 10–6 and Example 10–7.

Example 10–5 Enqueuing time on the Producer

```
Producer myProducer(conn);
Message m(env);

// bytes is a Bytes object representing content of the message
m.setBytes(bytes);
myProducer.send( m, "QueueNmae");
```

Example 10–6 Dequeuing time on the Consumer

```
Consumer myConsumer(conn);

// Name must be registered with the queue through administrative interface
myConsumer.setConsumerName("ConsumerOne");
Message m = myConsumer.receive(Message::RAW);
Bytes b = m.getBytes();
```

Example 10–7 Setting the Agent on the Consumer

```
Consumer myConsumer(conn);
...
myConsumer.setAgent(ag);
myConsumer.receive();
```

Table 10–11 Constants of the Consumer Class

Constant	Description
DEQ_BROWSE	Read the message without acquiring a lock; equivalent to a SELECT.
DEQ_LOCKED	Read the message and get its write lock, which lasts s for the duration of the transaction; equivalent to a SELECT FOR UPDATE.
DEQ_REMOVE	Default. Read the message, and update or delete it; the message can be retained in the queue table based on the retention properties.

Table 10–11 (Cont.) Constants of the Consumer Class

Constant	Description
DEQ_NO_DATA	Confirm receipt of the message, but do not deliver its actual content.
DEQ_FIRST_MSG	Retrieves the first available message on the queue that matches the search criteria; resets the position to the beginning of the queue.
DEQ_NEXT_MSG	Default. Retrieves the next available message on the queue that matches the search criteria. If the previous message belongs to a message group, AQ will retrieve the next available message that matches the search criteria and belongs to the message group.
DEQ_NEXT_ TRANSACTION	Skips the remainder of the current transaction group, if any, and retrieves the first message of the next transaction group. This option can only be used if message grouping is enabled for the current queue.
DEQ_ON_COMMIT	Default. The dequeue will be part of the current transaction.
DEQ_IMMEDIATE	The dequeued message is not part of the current transaction. It constitutes a transaction on its own.
DEQ_NO_WAIT	Do not wait if there are no messages on the queue.
DEQ_WAIT_FOREVER	The consumer will wait for the `Message` indefinitely

Table 10–12 Summary of Consumer Methods

Method	Description
Consumer() on page 10-80	`Consumer` class constructor.
getConsumerName() on page 10-81	Retrieves the name of the `Consumer`.
getCorrelationId() on page 10-81	Retrieve she correlation id of the message that is to be dequeued.
getDequeueMode() on page 10-81	Retrieves the dequeue mode of the `Consumer`.
getMessageIdToDequeue() on page 10-82	Retrieves the id of the message that will be dequeued.
getQueueName() on page 10-82	Gets the name of the queue used by the consumer.
getPositionOfMessage() on page 10-82	Retrieves the position of the Message that will be dequeued.
getTransformation() on page 10-82	Retrieves the transformation applied before a `Message` is dequeued.

Table 10–12 (Cont.) Summary of Consumer Methods

Method	Description
getVisibility() on page 10-83	Retrieves the transactional behavior of the dequeue operation.
getWaitTime() on page 10-83	Retrieve the specified behavior of the `Consumer` when waiting for a `Message` with matching search criteria.
isNull() on page 10-83	Tests whether the `Consumer` object is `NULL`.
receive() on page 10-83	Receive and dequeue a `Message`
setAgent() on page 10-84	Sets the `Agent`'s name and address (queue name) on the consumer.
setConsumerName() on page 10-84	Sets the `Consumer` name.
setCorrelationId() on page 10-85	Specify the correlation identifier of the message to be dequeued.
setDequeueMode() on page 10-85	Specify the locking behavior associated with dequeuing.
setMessageIdToDequeue() on page 10-85	Specifies the identifier of the `Message` to be dequeued.
setNull() on page 10-86	Nullifies the `Consumer`; frees the memory associated with this object.
setPositionOfMessage() on page 10-86	Specifies position of the `Message` to be retrieved.
setQueueName() on page 10-86	Specifies the name of a queue prior to dequeuing `Messages`.
setTransformation() on page 10-87	Specify transformation applied before dequeuing a `Message`.
setVisibility() on page 10-87	Specify if `Message` should be dequeued as part of the current transaction.
setWaitTime() on page 10-87	Specify wait conditions if there are no `Messages` with matching criteria.

Consumer()

Consumer class constructor.

Syntax	Description

`Consumer(`
`   Connection *conn);`
	Creates a new `Consumer` object with the specified `Connection` handle.

`Consumer(`
`   Connection *conn`
`   Agent agent);`
	Creates a new `Consumer` object with specified `Connection` handle and properties of the specified `Agent`.

`Consumer(`
`   Connection *conn,`
`   const string& queue);`
	Creates a new `Consumer` object with specified `Connection` handle and queue.

Parameter	Description
`conn`	The connection in which the `Consumer` is created.
`queue`	Queue at which the `Consumer` retrieves messages.

getConsumerName()

Retrieve the name of the `Consumer`.

Syntax

```
string getConsumerName() const;
```

getCorrelationId()

Retrieve she correlation id of the message that is to be dequeued

Syntax

```
string geCorrelationId() const;
```

getDequeueMode()

Retrieve the dequeue mode of the `Consumer`. These valid values for the dequeue mode are defined as constants of the `Consumer` class in Table 10–11 on page 10-78: `DEQ_BROWSE`, `DEQ_LOCKED`, `DEQ_REMOVE` and `DEQ_REMOVE_NODATA`.

Syntax

```
DequeueMode getDequeueMode() const;
```

getMessageIdToDequeue()

Retrieves the id of the message that will be dequeued.

Syntax

```
Bytes getMessageToDequeue() const;
```

getPositionOfMessage()

Retrieves the position of the message that will be dequeued. These valid values for the position of the message are defined as constants of the `Consumer` class in Table 10–11 on page 10-78: `DEQ_FIRST_MSG`, `DEQ_NEXT_MSG` and `DEQ_NEXT_TRANSACTION`.

Syntax

```
Navigation getPositionOfMessage() const;
```

getQueueName()

Gets the name of the queue used by the consumer.

Syntax

```
string getQueueName() const;
```

getTransformation()

Retrieves the transformation applied before a `Message` is dequeued.

Syntax

```
string getTransformation() const;
```

getVisibility()

Retrieves the transactional behavior of the dequeue operation. These valid values for visibility are defined as constants of the `Consumer` class in Table 10–11 on page 10-78: `DEQ_ON_COMMIT` and `DEQ_IMMEDIATE`.

Syntax

```
Visibility getVisibility() const;
```

getWaitTime()

Retrieve the specified behavior of the `Consumer` when waiting for a `Message` with matching search criteria. These valid values for visibility are defined as constants of the `Consumer` class in Table 10–11 on page 10-78: `DEQ_WAIT_FOREVER` and `DEQ_NO_WAIT`.

Syntax

```
unsigned int getWaitTime() const;
```

isNull()

Tests whether the `Consumer` object is `NULL`. If the `Consumer` object is `NULL`, `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

receive()

Receives and dequeue a `Message`.

Syntax

```
Message receive(
   Message::PayloadType pType,
   const string& type="",
   const string& schema="");
```

Parameter	Description
pType	The type of payload expected. Payload Type is defined as: `enum PayloadType {RAW, ANYDATA, OBJECT};`
type	The type of the payload; considered only when pType is OBJECT.
schema	The schema in which the type is defined; considered only when pType is OBJECT.

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.

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.

setCorrelationId()

Specify 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`.

setDequeueMode()

Specify the locking behavior associated with dequeuing.

Syntax

```
void setDequeueMode(
   DequeueMode deqMode);
```

Parameter	Description
`deqMode`	Behavior of enqueuing. Valid values for the position of the message are defined as constants of the `Consumer` class in Table 10–11 on page 10-78: `DEQ_BROWSE`, `DEQ_LOCKED`, `DEQ_REMOVE` and `DEQ_REMOVE_NODATA`.

setMessageIdToDequeue()

Specifies the identifier of the `Message` to be dequeued.

Syntax

```
void setMessageIdToDequeue(
   const Bytes& id);
```

Parameter	Description
`id`	Identifier of the `Message` to be dequeued.

setNull()

Nullifies the `Consumer`; frees the memory associated with this object.

Syntax

```
void setNull();
```

setPositionOfMessage()

Specifies position of the `Message` to be retrieved.

Syntax

```
void setPositionOfMessage(
   Navigation nav);
```

Parameter	Description
`nav`	Position of the message. Valid values are defined as constants of the `Consumer` class in Table 10–11 on page 10-78: `DEQ_FIRST_MSG`, `DEQ_NEXT_MSG` and `DEQ_NEXT_TRANSACTION`.

setQueueName()

Specifies the name of a queue prior to dequeuing `Messages`. Typically used when dequeuing multiple messages from the same queue.

Syntax

```
void setQueueName(
   const string& queueName);
```

Parameter	Description
queueName	The name of a valid queue in the database.

setTransformation()

Specifies transformation applied before dequeuing the Message.

Syntax

```
void setTransformation(
   string &transFunction);
```

Parameter	Description
transFunction	SQL transformation function.

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 vis);
```

Parameter	Description
vis	Visibility option being set. Valid values are defined as constants of the Consumer class in Table 10–11 on page 10-78: DEQ_ON_COMMIT and DEQ_IMMEDIATE.

setWaitTime()

Specify wait conditions if there are no Messages with matching criteria. The waitTime parameter is ignored if messages in the same group are being dequeued.

Syntax

```
void setWaitTime(
   unsigned int waitTime);
```

Parameter	Description
`vwaitTime`	Waiting conditions. Valid values are defined as constants of the `Consumer` class in Table 10–11 on page 10-78: `DEQ_WAIT_FOREVER` and `DEQ_NO_WAIT`.

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 SQL92 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 numerical computations and also used to fetch from and set to the database.

The following code 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 = env->createConnection(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 10–13 Summary of Date Methods

Method	**Summary**
Date() on page 10-91	`Date` class constructor.
addDays() on page 10-91	Return a `Date` object with *n* days added.
addMonths() on page 10-92	Return a `Date` object with *n* months added.
daysBetween() on page 10-92	Return the number of days between the current `Date` object and the date specified.
fromBytes() on page 10-92	Convert an external `Bytes` representation of a `Date` object to a `Date` object.
fromText() on page 10-93	Convert the date from a given input string with format and nls parameters specified.
getDate() on page 10-94()	Return the date and time components of the `Date` object.
getSystemDate() on page 10-94	Return a `Date` object containing the system date.
isNull() on page 10-95	Returns `TRUE` if `Date` is `NULL; otherwise returns false`.
lastDay() on page 10-95	Returns a `Date` that is the last day of the month.
nextDay() on page 10-95	Returns a `Date` that is the date of the next day of the week.
operator=() on page 10-96	Assign the values of a date to another.
operator==() on page 10-96	Returns `TRUE` if `a` and `b` are the same, `false` otherwise.
operator!=() on page 10-96	Returns `TRUE` if `a` and `b` are unequal, `false` otherwise.
operator>() on page 10-97	Returns `TRUE` if `a` is past `b`, `false` otherwise.
operator>=() on page 10-97	Returns `TRUE` if `a` is past `b or equal to b`, `false` otherwise.
operator=() on page 10-96	Returns `TRUE` if `a` is before `b`, `false` otherwise.
operator>() on page 10-97	Returns `TRUE` if `a` is before `b`, or equal to `b`, `false` otherwise.
setDate() on page 10-99	Set the date from the date components input.
setNull() on page 10-99	Sets the object state to `NULL`.
toBytes() on page 10-99	Converts the `Date` object into an external `Bytes` representation.

Table 10–13 (Cont.) Summary of Date Methods

Method	Summary
toText() on page 10-100	Get the `Date` object as a string.
toZone() on page 10-100	Return a `Date` object converted from one time zone to another.

Date()

`Date` class constructor.

Syntax	Description
`Date();`	Creates a `NULL` `Date` object.

`Date(`
`   const Date &srcDate);`
	Create 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);`
	Create a `Date` object using integer parameters.

Parameter	Description
year	-4712 to 9999, except 0
month	1 to 12
day	1 to 31
minutes	0 to 59
seconds	0 to 59

addDays()

Adds a specified number of days to the `Date` object and returns the new date.

Syntax

```
Date addDays(
   int val) const;
```

Parameter	Description
`val`	The number of days to be added to the current `Date` object.

addMonths()

Adds a specified number of months to the `Date` object and returns the new date.

Syntax

```
Date addMonths(
   int val) const;
```

Parameter	Description
`val`	The number of months to be added to the current `Date` object.

daysBetween()

Returns the number of days between the current `Date` object and `the date specified`.

Syntax

```
IntervalDS daysBetween(
   const Date &date) const;
```

Parameter	Description
`date`	The date to be used to compute the days between.

fromBytes()

Converts a `Bytes` object to a `Date` object.

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.

fromText()

Set `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 Reference* for information on `TO_DATE`

Syntax	Description

`void fromText(`
`   const string &datestr,`
`   const string &fmt = "",`
`   const string &nlsParam = "",`
`   const Environment *envp = NULL);`
	Set `Date` object to value represented by a `string`.

`void fromText(`
`   const UString &datestr,`
`   const UString &fmt,`
`   const UString &nlsParam,`
`   const Environment *envp=NULL);`
	Set `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`.

Parameter	Description
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.

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 &sec) 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.

getSystemDate()

Returns the system date.

Syntax

```
static Date getSystemDate(
   const Environment *envP);
```

Parameter	Description
`envP`	The environment in which the system date is returned.

isNull()

Tests whether the `Date` is `NULL`. If the `Date` is `NULL`, `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

lastDay()

Returns a date representing the last day of the current month.

Syntax

```
Date lastDay() const;
```

nextDay()

Return a date representing the day after the day of the week specified.

See Also: *Oracle Database SQL Reference* for information on `TO_DATE`

Syntax	Description

`Date nextDay(`
`   const string &dow) const;`
	Return a date representing the day after the day of the week specified.

`Date nextDay(`
`   const UString &dow) const;`
	Return 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.

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.

operator==()

Compares the dates specified. If the dates are equal, 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.

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.

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` not the same 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.

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 not the same 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.

operator<()

Compares the dates specified. If the first date is in the past 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 not the same 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.

operator<=()

Compares the dates specified. If the first date is in the past 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 not the same 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.

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.

setNull()

Sets the `Date` to atomically `NULL`.

Syntax

```
void setNull();
```

toBytes()

Returns the date in `Bytes` representation.

Syntax

```
Bytes toBytes() const;
```

toText()

Return a `string` or `UString` with the value of this date formatted using `fmt` and `nlsParam`.

The value is interpreted based on the `fmt` and `nlsParam` parameters. In cases where `nlsParam` is not passed, the Globalization Support settings of the `envp` parameter are used.

> **See Also:** *Oracle Database SQL Reference* for information on `TO_DATE`

Syntax	Description

`void toText(`
`   const string &fmt = "",`
`   const string &nlsParam = "") const;`
	Return a `string` with the value of this date formatted using `fmt` and `nlsParam`.

`void toText(`
`   const UString &fmt,`
`   const UString &nlsParam) const;`
	Return 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`.

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
`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

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 10–14 Constants of the Environment Class

Constant	Description
`DEFAULT`	The default mode for creating an `Environment` object; no thread safety or object support
`OBJECT`	Mode for creating an `Environment` object; uses object features.
`SHARED`	Shared mode for creating an `Environment` object.
`NO_USERCALLBACKS`	Mode for creating an `Environment` object; does not support user callbacks.
`THREADED_MUTEXED`	Thread safe mode for creating an `Environment` object, mutexed internally by OCCI
`THREADED_UNMUTEXED`	Thread safe mode for creating an `Environment` object, client is responsible for mutexing.
`EVENTS`	Mode for creating an `Environment` object; supports registration for event notification (Advanced Queuing).
`USE_LDAP`	Mode for creating an `Environment` object; supports registration with LDAP.

Table 10–15 Summary of Environment Methods

Method	Summary
createConnection() on page 10-104	Establish a connection to the specified database.
createConnectionPool() on page 10-105	Create a connection pool.
createEnvironment() `on page 10-106`	Create an `Environment` object.
createStatelessConnectionPool() `on page 10-107`	Create a stateless connection pool.

Table 10–15 (Cont.) Summary of Environment Methods

Method	Summary
enableSubscription() on page 10-108	Enables subscription notification
disableSubscription() on page 10-109	Disables subscription notification
getCacheMaxSize() on page 10-109	Retrieve the Cache Max heap size.
getCacheOptSize() on page 10-109	Retrieve the cache optimal heap size.
getCacheSortedFlush() on page 10-109	Retrieve the setting of the cache sorting flag.
getCurrentHeapSize() on page 10-110	Return the current amount of memory allocated to all objects in the current environment.
getEnvironment() on page 10-110	Returns the `Environment` used for creating a `Subscription`.
getLDAPAdminContext() on page 10-110	Returns the administrative context when using LDAP open notification registration.
getLDAPAuthentication() on page 10-110	Returns the authentication mode when using LDAP open notification registration.
getLDAPHost() on page 10-110	Returns the host on which the LDAP server runs.
getLDAPPort() on page 10-110	Returns the port on which the LDAP server is listening.
getMap() on page 10-111()	Return the Map for the current environment.
getOCIEnvironment() on page 10-111	Return the OCI environment associated with the current environment.
getXAConnection() on page 10-111	Create an XA connection to a database.
getXAEnvironment() on page 10-112	Create an XA `Environment` object.
releaseXAConnection() on page 10-112	Release all resources allocated by a getXAConnection() call.
releaseXAEnvironment() on page 10-112	Release all resources allocated by a getXAEnvironment() call.
setCacheMaxSize() on page 10-113	Specifies the maximum size for the client-side object cache as a percentage of the optimal size.
setCacheOptSize() on page 10-113	Specifies the optimal size for the client-side object cache in bytes.

Table 10–15 (Cont.) Summary of Environment Methods

Method	Summary
setCacheSortedFlush() on page 10-113	Speechifies whether to sort cache in table order prior to flushing.
setLDAPAdminContext() on page 10-114	Speechifies the administrative context for the LDAP client.
setLDAPAuthentication() on page 10-114	Speechifies the LDAP authentication mode.
setLDAPHostAndPort() on page 10-115	Speechifies the LDAP server host and port.
setLDAPLoginNameAndPassword() on page 10-115	Speechifies the login name and password when connecting to an LDAP server.
terminateConnection() on page 10-115	Close the connection pool and free all related resources.
terminateConnectionPool() `on page 10-116`	Close the connection pool and free all related resources.
terminateEnvironment() `on page 10-116`	Destroy the environment.
terminateStatelessConnectionPool() `on page 10-116`	Close the stateless connection pool and free all related resources.

createConnection()

This method establishes a connection to the database specified.

Syntax	Description

`Connection * createConnection(`
`   const string &username,`
`   const string &password,`
`   const string &connectString);`
	Creates a default connection.

`Connection * createConnection(`
`   const UString &username,`
`   const UString &password,`
`   const UString &connectString);`
	Creates a connection (Unicode support). The client `Environment` should be initialized in `OCCIUTIF16` mode.

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.

createConnectionPool()

Create 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);`
	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);`
	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.

Parameter	Description
maxConn	The maximum number of connections in the pool. Valid values are 1 and greater.
incrConn	The increment by which to increase the number of connections to be opened if the current number of connections is less than maxConn. Valid values are 1 and greater.

createEnvironment()

Create 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 specified is THREADED_MUTEXED or THREADED_UNMUTEXED, 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.

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,`
`   const std::string &charset,`
`   const std::string &ncharset);`
	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	Valid values are (see Table 10–14 on page 102 for descriptions) DEFAULT, THREADED_MUTEXED, THREADED_UNMUTEXED, OBJECT.
ctxp	Context pointer for user-defined memory management function.
size	The size of the memory to be allocated by user-defined memory allocation function.
newsize	The new size of the memory to be reallocated.
memptr	the existing memory that needs to 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 will replace the one specified in NLS_LANG.
ncharset	Character set id that will replace the one specified in NLS_NCHAR.

createStatelessConnectionPool()

Create a StatelessConnectionPool object with specified pool attributes.

Syntax

```
StatelessConnectionPool* createStatelessConnectionPool(
   const string &poolUserName,
   const string &poolPassword,
   const string connectString="",
   unsigned int maxConn=1,
   unsigned int minConn=0,
   unsigned int incrConn=1,
   enum StatelessConnectionPool::Mode=StatelessConnectionPool::HETEROGENEOUS);
```

Parameter	Description
`poolUserName`	The pool user name.
`poolPassword`	The pool password.
`connectString`	The connection string for the server
`maxConn`	The maximum number of connections that can be opened in the pool; additional sessions cannot be open.
`minConn`	The number of connections initially created in a pool. This parameter is considered only if the mode is set to `HOMOGENEOUS`.
`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 mode is set to `HOMOGENEOUS`.
`Mode`	The mode of the connection pool, which can be either `HIOMOGENEOUS` or `HETEROGENEOUS`. ■ `HETEROGENEOUS` -- default mode; connections with different authentication contexts can be created in the same pool. ■ `HOMOGENEOUS` -- all connections in the pool will be 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.

enableSubscription()

Enables subscription notification.

Syntax

```
void enableSubscription(
   Subscription &subscr);
```

Parameter	Description
subscr	The Subscription.

disableSubscription()

Disables subscription notification.

Syntax

```
void disableSubscription(
   Subscription &subscr);
```

Parameter	Description
subscr	The Subscription.

getCacheMaxSize()

Retrieves the Cache Max heap size.

Syntax

```
unsigned int getCacheMaxSize() const;
```

getCacheOptSize()

Retrieves the Cache optimal heap size.

Syntax

```
unsigned int getCacheOptSize() const;
```

getCacheSortedFlush()

Retrieves the current setting of the cache sorting flag; `TRUE` or `FALSE`.

Syntax

```
bool getCacheSortedFlush();
```

getCurrentHeapSize()

Returns the amount of memory currently allocated to all objects in this environment.

Syntax

```
unsigned int getCurrentHeapSize() const;
```

getEnvironment()

Returns the `Environment` used for creating a `Subscription`.

Syntax

```
Environment* getEnvironment() const;
```

getLDAPAdminContext()

Returns the administrative context when using LDAP open notification registration.

Syntax

```
string getLDAPAdminContext() const;
```

getLDAPAuthentication()

Returns the authentication mode when using LDAP open notification registration.

Syntax

```
unsigned int getLDAPAuthentication() const;
```

getLDAPHost()

Returns the host on which the LDAP server runs.

Syntax

```
string getLDAPHost() const;
```

getLDAPPort()

Returns the port on which the LDAP server is listening.

Syntax

```
string unsigned intgetLDAPPort() const;
```

getMap()

Returns a pointer to the map for this environment.

Syntax

```
Map *getMap() const;
```

getOCIEnvironment()

Returns a pointer to the OCI environment associated with this environment.

Syntax

```
LNOCIEnv *getOCIEnvironment() const;
```

getXAConnection()

Returns a pointer to an OCCI Connection object that corresponds to the one opened by the XA library.

Syntax

```
get XAConnection(
   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).

getXAEnvironment()

Returns a pointer to an OCCI Environment object that corresponds to the one opened by the XA library.

Syntax

```
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).

releaseXAConnection()

Release/deallocate all resources allocated by the getXAConnection() method.

Syntax

```
releaseXAConnection(
   Connection* conn);
```

Parameter	Description
conn	The connection returned by the getXAConnection() method.

releaseXAEnvironment()

Release/deallocate all resources allocated by the getXAEnvironment() method.

Syntax

```
releaseXAEnvironment(
   Environment* env);
```

Parameter	Description
`env`	The environment returned by the getXAEnvironment() method.

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.

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.

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 prior to flushing from client cache.

Syntax

```
void setCacheSortedFlush(
   bool flag);
```

Parameter	Description
`flag`	`FALSE` (default) -- no sorting; `TRUE` -- sorting in table order

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

setLDAPAuthentication()

Specifies the authentication mode. Currently the only supported value is 0x1: Simple authentication; username/password authentication.

Syntax

```
void setLDAPAuthentication(
   unsigned int mode);
```

Parameter	Description
`mode`	The authentication mode

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

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

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.

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.

terminateEnvironment()

Closes the environment, and frees all related system resources.

Syntax

```
void terminateEnvironment(
   Environment *env);
```

Parameter	Description
`env`	Environment to be closed.

terminateStatelessConnectionPool()

Destroy the specified Stateless Connection Pool.

Syntax

```
void termimnateStatelessConnectionPool(
StatelessConnectionPool* scp,
StatelessConnectionPool::DestroyMode mode=StatelessConnectionPool::DEFAULT);
```

Parameter	Description
`scp`	The `Stateless ConnectionPool` to be destroyed.
mode	Valid values are: ■ `StatelessConnectionPool::DEFAULT` -- if there are still active busy connections in the pool, ORA24422 error is thrown. ■ `StatelessConnectionPool::SPD_FORCE` -- any busy connections in the pool will be forcefully terminated and the pool destroyed.

Warning: When the pool is terminated using the `StatelessConnectionPool::SPD_FORCE` mode, the user will lose memory corresponding to the number of connections forcefully terminated.

IntervalDS Class

Leading field precision will be determined by number of decimal digits in day input. Fraction second precision will be determined by number of fraction digits on input.

```
IntervalDS(
   const Environment *env,
   int day = 0,
   int hour = 0,
   int minute = 0,
   int second = 0,
   int fs = 0);
```

Table 10–16 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().

The following code example demonstrates that 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
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();
```

The following code 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 10–17 Summary of IntervalDS Methods

Method	Summary
IntervalDS() on page 10-120	`IntervalDS` class constructor.
fromText() on page 10-121	Return an `IntervalDS` with the value represented by `instring`.
getDay() on page 10-121	Return day interval values.
getFracSec() on page 10-122	Return fractional second interval values.
getFracSec() on page 10-122	Return hour interval values.
getMinute() on page 10-122	Return minute interval values.
getSecond() on page 10-122	Return second interval values.

Table 10–17 (Cont.) Summary of IntervalDS Methods

Method	Summary
isNull() on page 10-122	Return `true` if `IntervalDS` is `NULL`, `false` otherwise.
operator*() on page 10-123	Return the product of two `IntervalDS` values.
operator*=() on page 10-123	Multiplication assignment.
operator=() on page 10-31	Simple assignment.
operator==() on page 10-124	Check if `a` and `b` are equal.
operator!=() on page 10-124	Check if `a` and `b` are not equal.
operator==() on page 10-124	Return an `IntervalDS` with value `(a / b).`
operator/=() on page 10-125	Division assignment.
operator>() on page 10-125	Check if `a` is greater than `b`
operator>=() on page 10-97	Check if `a` is greater than or equal to `b`.
operator<() on page 10-126	Check if `a` is less than `b`.
operator>() on page 10-97	Check if `a` is less than or equal to `b`.
operator>() on page 10-125	Return an `IntervalDS` with value (`a - b`).
operator-=() on page 10-127	Subtraction assignment.
operator+() on page 10-128	Return the sum of two `IntervalDS` values.
operator+=() on page 10-128	Addition assignment.
getDay() on page 10-121	Set day-second interval.
setName() on page 10-33	Set day-second interval to `NULL`.
operator=() on page 10-96	Return string representation for the interval.

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 IntervalDS &src);`
	Constructs an `IntervalYM` object from src.

Parameter	Description
`scp`	The source that the `IntervalDS` object will be copied from.

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 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 will be used.

getDay()

Returns the day component of the interval.

Syntax

```
int getDay() const;
```

getFracSec()

Returns the fractional second component of the interval.

Syntax

```
int getFracSec() const;
```

getHour()

Returns the hour component of the interval.

Syntax

```
int getHour() const;
```

getMinute()

Returns the minute component of this interval.

Syntax

```
int getMinute() const;
```

getSecond()

Returns the seconds component of this interval.

Syntax

```
int getSecond() const;
```

isNull()

Tests whether the interval is `NULL`. If the interval is `NULL` then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

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.

operator*=()

Assigns the product of `IntervalDS` and `a` to `IntervalDS`.

Syntax

```
IntervalDS& operator*=(
   const IntervalDS &val);
```

Parameter	Description
`val`	A day second interval.

operator=()

Assigns the specified value to the interval.

Syntax

```
IntervalDS& operator=(
   const IntervalDS &val);
```

Parameter	Description
`val`	Value to be assigned.

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.

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.

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.

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.

operator/=()

Assigns the quotient of `IntervalDS` and `val` to `IntervalDS`.

Syntax

```
IntervalDS& operator/=(
   const IntervalDS &val);
```

Parameter	Description
`val`	A day second interval.

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.

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.

operator<()

Compares the intervals specified. If the first interval is less than the second interval then `TRUE` is returned; otherwise, `FALSE` is returned. If either interval is `NULL` then SQLException is thrown.

Syntax

```
bool operator<(
   const IntervalDS &first,
   const IntervalDS &second);
```

Parameter	Description
`first`	The first interval to be compared.
`second`	The second interval to be compared.

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.

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.

operator-=()

Assigns the difference between `IntervalDS` and `val` to `IntervalDS`.

Syntax

```
IntervalDS& operator-=(
   const IntervalDS &val);
```

Parameter	Description
`val`	A day second interval.

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.

operator+=()

Assigns the sum of `IntervalDS` and `val` to `IntervalDS`.

Syntax

```
IntervalDS& operator+=(
   const IntervalDS &val);
```

Parameter	Description
`val`	A day second interval.

set()

Sets the interval to the values specified.

Syntax

```
void set(
   int day,
   int hour,
   int minute,
   int second,
   int fracsec);
```

Parameter	Description
`day`	Day component.
`hour`	Hour component.
`min`	Minute component.
`second`	Second component.
`fracsec`	Fractional second component.

setNull()

Sets the `IntervalDS` to `NULL`.

Syntax

```
void setNull();
```

toText()

Returns the 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.

Parameter	Description
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.

IntervalYM Class

IntervalYM supports the SQL92 datatype Year-Month Interval.

Leading field precision will be determined by number of decimal digits on input.

```
IntervalYM(
   const Environment *env,
   int year = 0,
   int month=0);
```

Table 10–18 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.

The following code 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
Interval UM anotherYM(env, "10-30");
ym=anotherYM;

// Now all operations on YM are valid
int yr = ym.getYear();
```

The following code 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

IntervalYM mulYm1 = ym1 * Number(env, 10);    //call * operator
if(ym1<mulYm1)    //comparison
   .
   .
string strym = ym1.toText(3);    //3 is the leading field precision
```

Table 10–19 Summary of IntervalYM Methods

Method	Summary
IntervalYM() on page 10-133	`IntervalYM` class constructor.
operator*() on page 10-123	Return an `IntervalYM` with the value represented by `instring`.
getMonth() on page 10-134	Return month interval value.
getYear() on page 10-134	Return year interval value.
isNull() on page 10-30	Check if the interval is `NULL`.
operator*() on page 10-123	Return the product of two `IntervalYM` values.
operator*=() on page 10-123	Multiplication assignment.
operator=() on page 10-31	Simple assignment.
operator+() on page 10-128	Check if `a` and `b` are equal.
operator!=() on page 10-124	Check if `a` and `b` are not equal.
operator==() on page 10-124	Return an `interval` with value `(a/b)`.
operator/=() on page 10-125	Division assignment.
operator>() on page 10-125	Check if `a` is greater than `b`.
operator>=() on page 10-97	Check if `a` is greater than or equal to `b`.
operator<() on page 10-126	Check if `a` is less than `b`.
operator>() on page 10-97	Check if `a` is less than or equal to `b`.
operator+() on page 10-128	Return an `interval` with value `(a - b)`.

Table 10–19 (Cont.) Summary of IntervalYM Methods

Method	Summary
operator-=() on page 10-139	Subtraction assignment.
operator=() on page 10-31	Return the sum of two `IntervalYM` values.
operator+=() on page 10-128	Addition assignment.
getDay() on page 10-121	Set the interval to the values specified.
setName() on page 10-33	Set the interval to `NULL`.
operator=() on page 10-96	Return the string representation of the interval.

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().

`IntervalDS(`
`   const IntervalYM &src);`
	Constructs an `IntervalYM` object from val.

Parameter	Description
`scp`	The source that the `IntervalYM` object will be copied from.

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 will be used.

getMonth()

This method returns the month component of the interval.

Syntax

```
int getMonth() const;
```

getYear()

This method returns the year component of the interval.

Syntax

```
int getYear() const;
```

isNull()

This method tests whether the interval is `NULL`. If the interval is `NULL` then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

operator*()

This method multiplies the interval by a factor and returns the result.

Syntax

```
const IntervalYM operator*(
   const IntervalDS &interval
   const Number &value);
```

Parameter	Description
`interval`	Interval to be multiplied.
`val`	Value by which interval is to be multiplied.

operator*=()

This method multiplies the interval by a specified value.

Syntax

```
IntervalYM& operator*=(
   const Number &val);
```

Parameter	Description
`val`	Value to be multiplied.

operator=()

This method assigns the specified value to the interval.

Syntax

```
const IntervalYM& operator=(
   const IntervalYM &val);
```

Parameter	Description
`val`	Value to be assigned.

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.

operator!=()

This method compares the intervals specified. If the intervals are not equal then `TRUE` is returned; otherwise, `FALSE` is returned. If either interval is `NULL` then SQLException is thrown.

Syntax

```
bool operator!=(
   const IntervalYM &first,
   const IntervalYM &second);
```

Parameter	Description
`first`	The first interval to be compared.
`second`	The second interval to be compared.

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.

operator/=()

This method divides the interval by a factor.

Syntax

```
IntervalYM& operator/=(
   const Number &val);
```

Parameter	Description
`val`	A day second interval.

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);
```

Parameter	Description
`first`	The first interval to be compared.
`second`	The second interval to be compared.

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.

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.

operator<=()

This method compares the intervals specified. If the first interval is less than or equal to the second interval then `TRUE` is returned; otherwise, `FALSE` is returned. If either interval is `NULL` then SQLException is thrown

Syntax

```
bool operator<=(
   const IntervalYM &first,
   const IntervalYM &second);
```

Parameter	Description
first	The first interval to be compared.
second	The second interval to be compared.

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.

operator-=()

This method computes the difference between itself and another interval.

Syntax

```
IntervalYM& operator-=(
   const IntervalYM &val);
```

Parameter	Description
val	A day second interval.

operator+()

This method returns the sum of 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.

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.

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`.

setNull()

This method sets the interval to `NULL`.

Syntax

```
void setNull();
```

toText()

This method returns the string representation of the interval.

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`.

Listener Class

The `Listener` class encapsulates the ability to listen for `Messages`, on behalf of registered `Agents`, at specified queues.

Table 10–20 Summary of Listener Methods

Method	Summary
Listener() on page 10-142	`Listener` class constructor.
getAgentList() on page 10-143	Retrieve the list of `Agents` for which the `Listener` provides its services.
getTimeOutForListen() on page 10-143	Retrieve the time out for a call.
listen() on page 10-143	Listens for `Messages` and returns the name of the `Agent` for whom a `Message` is intended.
setAgentList() on page 10-144	Specifies the list of `Agents` for which the `Listener` provides its services.
setTimeOutForListen() on page 10-144	Specifies the time out for a listen() call.

Listener()

`Listener` class constructor.

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.

Parameter	Description
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.

getAgentList()

Retrieve the list of Agents for which the Listener provides its services.

Syntax

```
vector<Agent> getAgentList() const;
```

getTimeOutForListen()

Retrieve the time out for a call.

Syntax

```
int getTimeOutForListen() const;
```

listen()

Listens for Messages on behalf of specified Agents for the amount of time specified by a previous setTimeOutForListen() call. Returns the Agent for which there is a Message.

Syntax

```
Agent listen();
```

Note: This is a blocking call. Prior to this call, the following steps should be completed:

- Each Agent must be registered with the listener using setAgentList() method.
- The time out parameter should be set using the setTimeOutForListen() method. This is a blocking call 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.

setAgentList()

Specifies the list of `Agents` for which the `Listener` provides its services.

Syntax

```
void setAgentList(
   vector<Agent>& list);
```

Parameter	Description
`list`	The list of `Agents`.

setTimeOutForListen()

Specifies the time out for a listen() call.

Syntax

```
void setTimeOutForListen(
   int timeOut);
```

Parameter	Description
`timeOut`	The time interval, in seconds, during which the `Listener` is waiting for `Messages` at specified queues.

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 need to customize. In addition, you must add an entry for each such class into the `Map` member of the `Environment`.

Table 10–21 Summary of MetaData Methods

Method	Summary
put() on page 10-145	Adds a map entry for the type to be customized.

put()

Adds a map entry for the type, `type_name`, that you want to customize; you must implement the `type_name` C++ class.

You must then add this information into a map object, which should be registered with the connection if the user wants the standard mappings to overridden.This registration can be done by calling the this method after the environment is created passing the environment.

Syntax	Description
`void put( const string &schemaType, void *(*rSQL)(void *), void (*wSQL) (void *, void *));`	Register a type and its corresponding C++ `readSQL` and `writeSQL` functions.

Syntax	Description

`void put(`
`   const std::string& schName,`
`   const std::string& typName,`
`   void *(*rSQL)(void *),`
`   void (*wSQL)(void *, void *));`
	Register 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 *));`
	Register a type and its corresponding C++ `readSQL` and `writeSQL` functions; unicode support.

Parameter	Description
`schemaType`	The schema and typename, separated by ".", like `SCOTT.TYPE1`
`schName`	Name of the scema
`typName`	Name of the type
`rDQL`	The `readSQL` function ponter of the C++ class that corresponds to the type
`wSQL`	The `writeSQL` function ponter of the C++ class that corresponds to the typ

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 10–22 Constants of the Message Class

Constant	Description
`MSG_WAITING`	The message delay time has not been reached.
`MSG_READY`	The message is ready to be processed.
`MSG_PROCESSED`	The message has been processed, and is being and is retained.
`MSG_EXPIRED`	The message has been moved to the exception queue.

Table 10–23 Summary of Message Methods

Method	Summary
Message() `on page 10-148`	`Message` class constructor.
getAnyData() `on page 10-149`	Retrieves `AnyData` payload of the message.
getAttemptsToDequeue() `on page 10-149`	Retrieves the number of attempts made to degueue the message.
getBytes() `on page 10-149`	Retrieves `Bytes` payload of the message.
getCorrelationId() `on page 10-149`	Retrieves the identification string.
getDelay() `on page 10-149`	Retrieves delay with which message was enqueued.
getExceptionQueueName() `on page 10-150`	Retrieves name of queue to which `Message` is moved when it cannot be processed.
getExpiration() `on page 10-150`	Retrieves the expiration of the message.
getMessageEnqueuedTime() `on page 10-150`	Retrieves time at which message was enqueued.
getMessageState() `on page 10-150`	Retrieves state of the message at time of enqueuing.
getObject() `on page 10-151`	Retrieves object payload of the message.
getPriority() `on page 10-151`	Retrieves the priority of the message.

Table 10–23 (Cont.) Summary of Message Methods

Method	Summary
getSenderId() on page 10-151	Retrieves the agent who send the `Message`.
isNull() on page 10-152	Tests whether the `Message` object is `NULL`.
setAnyData() on page 10-152	Specifies `AnyData` payload of the message.
setBytes() on page 10-152	Specifies `Bytes` payload of the message.
setCorrelationId() on page 10-153	Specifies the identification string.
setDelay() on page 10-153	Specifies the number of seconds to delay the enqueued `Message`.
setExceptionQueueName() on page 10-154	Specifies the name of the queue to which the `Message` object will be moved if it cannot be precessed.
setExpiration() on page 10-154	Specifies the duration of time that `Message` can be dequeued before it expires.
setNull() on page 10-155	Sets the `Message` object to `NULL`.
setObject() on page 10-155	Specifies object payload of the message.
setPreviousQueueId() on page 10-155	Specifies id of last queue that generated the `Message`.
setPriority() on page 10-156	Specifies priority of the `Message` object.
setRecipientList() on page 10-156	Specifies the list of agents for whom the message is intended.
setSenderId() on page 10-156	Specifies the sender of the `Message`.

Message()

Message class constructor.

Syntax

```
Message(
   Environment *env);
```

Parameter	Description
`env`	The environment of the `Message`.

getAnyData()

Retrieve the `AnyData` payload of the `Message`.

Syntax

```
AnyData getAnyData() const;
```

getAttemptsToDequeue()

Retrieves the number of attempts made to degueue the message. This property cannot be retrieved while enqueuing.

Syntax

```
int getAttemptsToDequeue();
```

getBytes()

Retrieve `Bytes` payload of the `Message`.

Syntax

```
Bytes getBytes() const;
```

getCorrelationId()

Retrieves the identification string.

Syntax

```
string getCorrelationId();
```

getDelay()

Retrieves the delay (in seconds) with which the `Message` was enqueued.

Syntax

```
int getDelay();
```

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();
```

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();
```

getMessageEnqueuedTime()

Retrieves the time at which the message was enqueued, in `Date` format. This value is determined by the system, and cannot be set by the user.

Syntax

```
Date getMessageEnqueuedTime() const;
```

getMessageState()

Retrieves the state of the message at the time of enqueuing. This parameter cannot be set an enqueuing time. The possible states are defined as constants of the `Message` class in Table 10–22 on page 10-147: `MSG_WAITING`, `MSG_READY`, `MSG_PROCESSED` and `MSG_EXPIRED`.

Syntax

```
MessageState getMessageState();
```

getObject()

Retrieve object payload of the `Message`.

Syntax

```
PObject* getObject() const;
```

getPreviousQueueId()

Retrieves id of the message in the last queue that generated the message.

Syntax

```
Bytes getPreviousQueueId();
```

getPriority()

Retrieves the priority of the `Message`.

Syntax

```
int getPriority();
```

getSenderId()

Retrieves the agent who send the `Message`.

Syntax

```
Agent getSenderId();
```

getType()

Retrieve the type and schema of the data in the `Message`.

Syntax

```
void getType(
   string& schema,
   string& type);
```

Parameter	Description
schema	Schema of the data.
type	Type of the.data.

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();
```

setAnyData()

Specifies `AnyData` payload of the `Message`.

Syntax

```
void setAnyData(
   const AnyData& anydata);
```

Parameter	Description
anydata	Data content of the `Message`.

setBytes()

Specifies `Bytes` payload of the `Message`.

Syntax

```
void setBytes(
   const Bytes& bytes);
```

Parameter	Description
bytes	Data content of the Message.

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.

setDelay()

Specifies the time (in seconds) to delay the enqueued Message. After the delay ends, the Message is available for dequeuing.

Note: Dequeuing by msgid overrides the delay specification. A Message enqueued with delay will be in the WAITING state. Delay is set by the producer of the Message.

Syntax

```
void setDelay(
   int delay);
```

Parameter	Description
delay	The delay.

setExceptionQueueName()

Specify the name of the queue to which the `Message` object will be moved if it cannot be processed successfully. The queue name must be valid.

Note:

- If the exception queue does not exist at the time of the move, the `Message` will be moved to the default exception queue associated with the queue table; a warning will be logged in the alert file.
- If the default exception queue is used, the parameter will return a `NULL` value at enqueuing time; the attribute must refer to a valid queue name.

Syntax

```
void setExceptionQueueName(
   const string& name);
```

Parameter	Description
`name`	The name of the exception queue.

setExpiration()

Specify the duration time (in seconds) that the `Message` object is available for dequeuing. A `Message` expires after the this time.

Syntax

```
void setExpiration(
   int expiration);
```

Parameter	Description
`expiration`	The duration of expiration.

setNull()

Sets the `Message` object to `NULL`. Before the `Connection` is destroyed by the terminateConnection() call of the Environment Class, all `Message` objects need to be set to `NULL`.

Syntax

```
void setNull();
```

setObject()

Specifies object payload of the `Message`.

Syntax

```
void setObject(
   const PObject& pobj);
```

Parameter	Description
`pobj`	Content of the data

setPreviousQueueId()

Specify the id of the last queue that generated the `Message` object.

Syntax

```
void setPreviousQueueId(
   Bytes& id);
```

Parameter	Description
`id`	The id of the queue.

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`.

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 will override the default subscriber list.

Syntax

```
void setRecipientList(
   vector<Agent>& agentList);
```

Parameter	Description
`agentList`	The list of `Agents`.

setSenderId()

Specifies the sender of the `Message`.

Syntax

```
void setSenderId(
   const Agent& id);
```

Parameter	Description
`id`	Sender id.

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 parameter types for objects are listed in Table 10–24.

Table 10–24 Parameter Types for Objects

Parameter Type	Description
PTYPE_ARG	argument of a function or procedure
PTYPE_COL	column of a table or view
PTYPE_DATABASE	database
PTYPE_FUNC	function
PTYPE_PKG	package
PTYPE_PROC	procedure
PTYPE_SCHEMA	schema
PTYPE_SEQ	sequence
PTYPE_SYN	synonym
PTYPE_TABLE	table
PTYPE_TYPE	type
PTYPE_TYPE_ARG	argument of a type method
PTYPE_TYPE_ATTR	attribute of a type
PTYPE_TYPE_COLL	collection type information
PTYPE_TYPE_METHOD	method of a type
PTYPE_TYPE_RESULT	results of a method
PTYPE_TYPE_UNK	object of an unknown type
PTYPE_TYPE_VIEW	view

Attribute values are returned on executing a `get`*`xxx`*`()` method, passing some attribute for which these are the results:

Table 10–25 Enumerated Values of Attributes for MetaData Class

Attribute Value	Description
`DURATION_SESSION`	Duration of a temporary table: session
`DURATION_TRANS`	Duration of a temporary table: transaction
`DURATION_NULL`	Duration of a temporary table: table not temporary
`TYPEENCAP_PRIVATE`	Encapsulation level of the method: private
`TYPEENCAP_PUBLIC`	Encapsulation level of the method: public
`TYPEPARAM_IN`	Argument mode: `IN`
`TYPEPARAM_OUT`	Argument mode: `OUT`
`TYPEPARAM_INOUT`	Argument mode: `IN/OUT`
`CURSOR_OPEN`	Effect of `COMMIT` operation on cursors and prepared statements in the database: preserve cursor state as before the `COMMIT` operation
`CURSER_CLOSED`	Effect of `COMMIT` operation on cursors and prepared statements in the database: cursors are closed on `COMMIT`, but the applicaton can still reexecute the statement without re-preparing it
`CL_START`	Position of the catalog in a qualified table: start
`CL_END`	Position of the catalog in a qualified table: end
`SP_SUPPORTED`	Database supports savepoints
`SP_UNSUPPORTED`	Database does not support savepoints
`NW_SUPPORTED`	Database supports nowait clause
`NW_UNSUPPORTED`	Database does not supports nowait clause
`AC_DDL`	Autocommit mode required for DDL statements
`NO_AC_DDL`	Autocommit mode not required for DDL statements
`LOCK_IMMEDIATE`	Locking mode for the database: immediate
`LOCK_DELAYED`	Locking mode for the database: delayed

Table 10–26 Summary of MetaData Methods

Method	Description
MetaData() on page 10-160	`MetaData` class constructor.
getAttributeCount() on page 10-160	Gets the count of the attribute as a `MetaData` object

Table 10–26 (Cont.) Summary of MetaData Methods

Method	Description
getAttributeId() on page 10-161	Gets the ID of the specified attribute
getAttributeType() on page 10-161	Gets the type of the specified attribute.
getBoolean() on page 10-161	Gets the value of the attribute as a C++ `boolean`.
getInt() on page 10-162	Gets the value of the attribute as a C++ `int`.
getMetaData() on page 10-162	Gets the value of the attribute as a `MetaData` object
getNumber() on page 10-162	Returns the specified attribute as a `Number` object.
getRef() on page 10-163	Gets the value of the attribute as a `Ref<T>`.
getString() on page 10-163	Gets the value of the attribute as a string.
getTimeStamp() on page 10-164	Gets the value of the attribute as a `Timestamp` object
getUInt() on page 10-164	Gets the value of the attribute as a C++ `unsigned int`.
getUString() on page 10-164	Return the value of the attribute as a `UString` in the character set associated with the metadata.
getVector() on page 10-165	Gets the value of the attribute as an C++ vector.
operator=() on page 10-165	Assigns one metadata object to another.

MetaData()

`MetaData` class constructor.

Syntax

```
MetaData(
   const MetaData &omd);
```

Parameter	Description
`cmd`	The source that the `MetaData` object will be copied from.

getAttributeCount()

This method returns the number of attributes related to the metadata object.

Syntax

```
unsigned int getAttributeCount() const;
```

getAttributeId()

This method returns the attribute ID (`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.

getAttributeType()

This method returns the attribute type (`NUMBER`, `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.

getBoolean()

This method returns the value of the attribute as a C++ `boolean`. If the value is a SQL `NULL`, the result is `FALSE`.

Syntax

```
bool getBoolean(
   MetaData::AttrId attributeId) const;
```

Parameter	Description
`attributeId`	The attribute ID

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

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

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

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

getString()

This method returns the value of the attribute as a string. If the value is SQL `NULL`, the result is `NULL`.

Syntax

```
string getString(
   MetaData::AttrId attributeId) const;
```

Parameter	Description
attributeId	The attribute ID

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

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

getUString()

Return 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

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

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

NotifyResult Class

A NotifyResult object holds the notification information in the Streams AQ notification callback. It is created by OCCI before invoking a user-callback, and is destroyed after the user-callback returns.

Table 10–27 Summary of NotifyResult Methods

Method	Summary
getConsumerName() on page 10-166	Returns the name of the notification consumer.
getMessage() on page 10-166	Returns the message.
getMessageId() on page 10-166	Returns the message ID.
getPayload() on page 10-167	Returns the payload.
getQueueName() on page 10-167	Returns the name of the queue.

getConsumerName()

Gets the name of the consumer for which the message has been enqueued. In case of a single consumer queue, this is a empty string.

```
string getConsumerName() const;
```

getMessage()

Gets the message which has been enqueued into the non-persistent queue.

```
Message getMessage() const;
```

getMessageId()

Gets the id of the message which has been enqueued.

```
Bytes getMessageId() const;
```

getPayload()

Gets the payload in case of a notification from `NS_ANONYMOUS` namespace.

```
Bytes getPayload() const;
```

getQueueName()

Gets the name of the queue on which the enqueue has happened

```
string getQueueName() const;
```

Number Class

The Number class handles limited-precision signed base 10 numbers. A Number guarantees 38 decimal digits of precision. All positive numbers in the range displayed here can be represented to a full 38-digit precision:

```
10^-130
```

and

```
9.99999999999999999999999999999999999999*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 numerical computations. They can also be used to fetch from and set to the database.

The following code 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);
Connection *conn = env->createConnection(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);

/* Execute 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 10–28 Summary of Number Methods

Method	Summary
Number() on page 10-172	`Number` class constructor.
abs() on page 10-173	Return the absolute value of the number.
arcCos() on page 10-174	Return the arcCosine of the number.
arcSin() on page 10-174	Return the arcSine of the number.
arcTan() on page 10-174	Return the arcTangent of the number.

Table 10–28 (Cont.) Summary of Number Methods

Method	Summary
arcTan2() on page 10-174	Return the arcTangent2 of the input number y and this number x.
ceil() on page 10-175	Return the smallest integral value not less than the value of the number.
cos() on page 10-175	Return the cosine of the number.
exp() on page 10-175	Return the natural exponent of the number.
floor() on page 10-175	Return the largest integral value not greater than the value of the number.
fromBytes() on page 10-176	Return a Number derived from a Bytes object.
fromText() on page 10-176	Return a Number from a given number string, format string and nls parameters specified.
hypCos() on page 10-177	Return the hyperbolic cosine of the number.
hypSin() on page 10-177	Return the hyperbolic sine of the number.
hypTan() on page 10-177	Return the hyperbolic tangent of the number.
intPower() on page 10-177	Return the number raised to the integer value specified.
isNull() on page 10-30	Check if Number is NULL.
ln() on page 10-178	Return the natural logarithm of the number.
log() on page 10-178	Return the logarithm of the number to the base value specified.
operator++() on page 10-179	Increment the number by 1.
operator--() on page 10-179	Decrement the number by 1.
operator*() on page 10-123	Return the product of two Numbers.
operator/() on page 10-180	Return the quotient of two Numbers.
operator%() on page 10-180	Return the modulo of two Numbers.
operator+() on page 10-181	Return the sum of two Numbers.
operator-() on page 10-181	Return the negated value of Number .
operator-() on page 10-181	Return the difference between two Numbers.
operator<() on page 10-182	Check if a number is less than an other number.
operator<=() on page 10-182	Check if a number is less than or equal to an other number.

Table 10–28 (Cont.) Summary of Number Methods

Method	**Summary**
operator>() on page 10-183	Check if a number is greater than an other number.
operator>=() on page 10-97	Check if a number is greater than or equal to an other number.
operator=() on page 10-96	Assign one number to another.
operator==() on page 10-96	Check if two numbers are equal.
operator!=() on page 10-96	Check if two numbers are not equal.
operator*=() on page 10-123	Multiplication assignment.
operator/=() on page 10-125	Division assignment.
operator%=() on page 10-186	Modulo assignment.
operator+=() on page 10-186	Addition assignment.
operator-=() on page 10-186	Subtraction assignment.
operator char() on page 10-187	Return `Number` converted to native char.
operator signed char() on page 10-187	Return `Number` converted to native signed char.
operator double() on page 10-187	Return `Number` converted to a native double.
operator float() on page 10-187	Return `Number` converted to a native float.
operator int() on page 10-188	Return `Number` converted to native integer.
operator long() on page 10-188	Return `Number` converted to native long.
operator long double() on page 10-188	Return `Number` converted to a native long double.
operator short() on page 10-188	Return `Number` converted to native short integer.
operator unsigned char() on page 10-188	Return `Number` converted to an unsigned native char.
operator unsigned int() on page 10-189	Return `Number` converted to an unsigned native integer.
operator unsigned long() on page 10-189	Return `Number` converted to an unsigned native long.
operator unsigned short() on page 10-189	Return `Number` converted to an unsigned native short integer.

Table 10–28 (Cont.) Summary of Number Methods

Method	Summary
power() on page 10-189	Return `Number` raised to the power of another number specified.
prec() on page 10-190	Return `Number` rounded to digits of precision specified.
round() on page 10-190	Return `Number` rounded to decimal place specified. Negative values are allowed.
setNull() on page 10-190	Set `Number` to `NULL`.
shift() on page 10-191	Return a `Number` that is equivalent to the passed value `* 10^n`, where `n` may be positive or negative.
sign() on page 10-191	Return 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() on page 10-191	Return sine of the number.
squareroot() on page 10-191	Return the square root of the number.
tan() on page 10-192	Returns tangent of the number.
toBytes() on page 10-192	Return a `Bytes` object representing the `Number`.
toText() on page 10-192	Return the number as a string formatted based on the format and nls parameters.
trunc() on page 10-193	Return a `Number` with the value truncated at `n` decimal place(s). Negative values are allowed.

Number()

`Number` class constructor.

Syntax	Description
`Number();`	Default constructor.

`Number(`
 `   const Number &val);`
	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`.

Syntax	Description

`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`.

`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
`val`	The value assigned to the `Number` object.

abs()

This method returns the absolute value of the `Number` object.

Syntax

```
const Number abs() const;
```

arcCos()

This method returns the arccosine of the `Number` object.

Syntax

```
const Numberconst Number arcCos() const;
```

arcSin()

This method returns the arcsine of the `Number` object.

Syntax

```
const Number arcSin() const;
```

arcTan()

This method returns the arctangent of the `Number` object.

Syntax

```
const Number arcTan() const;
```

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)`.

ceil()

This method returns the smallest integer that is greater than or equal to the `Number` object.

Syntax

```
const Number ceil() const;
```

cos()

This method returns the cosine of the `Number` object.

Syntax

```
const Number cos() const;
```

exp()

This method returns the natural exponential of the `Number` object.

Syntax

```
const Number exp() const;
```

floor()

This method returns the largest integer that is less than or equal to the `Number` object.

Syntax

```
const Number floor() const;
```

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.

fromText()

Set `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 Reference* for information on `TO_NUMBER`

Syntax	Description

`void fromText(`
`   const Environment *envp,`
`   const string &numstr,`
`   const string &fmt,`
`   const string &nlsParam = "");`
	Set `Number` object to value represented by a `string`.

`void fromText(`
`   const Environment *envp,`
`   const UString &numstr,`
`   const UString &fmt,`
`   const UString &nlsParam);`
	Set `Number` object to value represented by a `UString`.

Parameter	Description
`envp`	The OCCI environment.

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`.

hypCos()

This method returns the hypercosine of the `Number` object.

Syntax

```
const Number hypCos() const;
```

hypSin()

This method returns the hypersine of the `Number` object.

Syntax

```
const Number hypSin() const;
```

hypTan()

This method returns the hypertangent of the `Number` object.

Syntax

```
const Number hypTan() const;
```

intPower()

This method returns a `Number` whose value is the number object raised to the power of the value specified.

Syntax

```
const Number intPower(
   int val) const;
```

Parameter	Description
`val`	Power to which the number is raised.

isNull()

This method tests whether the `Number` object is `NULL`. If the `Number` object is `NULL`, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

ln()

This method returns the natural logarithm of the `Number` object.

Syntax

```
const Number ln() const;
```

log()

This method returns the logarithm of the `Number` object with the base provided by the parameter specified.

Syntax

```
const Number log(
   const Number &val) const;
```

Parameter	Description
`val`	The base to be used in the logarithm calculation.

operator++()

Unary `operator++()`. This method returns the `Number` object incremented by `1`. This is a prefix operator.

Syntax

```
Number& operator++();
```

operator++()

Unary `operator++()`. This method returns the `Number` object incremented by the integer specified. This is a postfix operator.

Syntax

```
const Number operator++(
   int incr);
```

Parameter	Description
`incr`	The number by which the `Number` object is incremented.

operator--()

Unary `operator--()`. This method returns the `Number` object decremented by `1`. This is a prefix operator.

Syntax

```
Number& operator--();
```

operator--()

Unary `operator--()`. This method returns the `Number` object decremented by the integer specified. This is a postfix operator.

Syntax

```
const Number operator--(
   int decr);
```

Parameter	Description
decr	The number by which the Number object is decremented.

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.

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.
divizor	The number by which to divide.

operator%()

This method returns the remainder of the division of the parameters specified.

Syntax

```
Number operator%(
   const Number &dividend,
   const Number &divider);
```

Parameter	Description
`dividend`	The number to be divided.
`divizor`	The number by which to divide.

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.

operator-()

Unary `operator-()`. This method returns the negated value of the `Number` object.

Syntax

```
const Number operator-();
```

operator-()

This method returns the difference between the parameters specified.

Syntax

```
Number operator-(
   const Number &subtrahend,
   const Number &subtractor);
```

Parameter	Description
`subtrahend`	The number to be reduced.
`subtractor`	The number to be subtracted.

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 is equal to 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.

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` is equal to 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.

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 is equal to 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.

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 is equal to infinity, then `FALSE` is returned.

Syntax

```
bool operator>=(
   const Number &first,
   const Number &second);
```

Parameter	Description
`first`	First number to be compared.

Parameter	Description
`second`	Second number to be compared.

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 is equal to +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.

operator!=()

This method checks whether the first parameter specified is equal to the second parameter specified. If the parameters are not equal, `TRUE` is returned; otherwise, `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.

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`.

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`.

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);
```

Parameter	Description
`num`	A parameter of type `Number`.

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`.

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`.

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.

operator char()

This method returns the value of the Number object converted to a native char.

Syntax

```
operator char() const;
```

operator signed char()

This method returns the value of the Number object converted to a native signed char.

Syntax

```
operator signed char() const;
```

operator double()

This method returns the value of the Number object converted to a native double.

Syntax

```
operator double() const;
```

operator float()

This method returns the value of the Number object converted to a native float.

Syntax

```
operator float() const;
```

operator int()

This method returns the value of the `Number` object converted to a native `int`.

Syntax

```
operator int() const;
```

operator long()

This method returns the value of the `Number` object converted to a native `long`.

Syntax

```
operator long() const;
```

operator long double()

This method returns the value of the `Number` object converted to a native `long double`.

Syntax

```
operator long double() const;
```

operator short()

This method returns the value of the `Number` object converted to a native `short` integer.

Syntax

```
operator short() const;
```

operator unsigned char()

This method returns the value of the `Number` object converted to a native `unsigned char`.

Syntax

```
operator unsigned char() const;
```

operator unsigned int()

This method returns the value of the `Number` object converted to a native `unsigned int`.

Syntax

```
operator unsigned int() const;
```

operator unsigned long()

This method returns the value of the `Number` object converted to a native `unsigned long`.

Syntax

```
operator unsigned long() const;
```

operator unsigned short()

This method returns the value of the `Number` object converted to a native `unsigned short` integer.

Syntax

```
operator unsigned short() const;
```

power()

This method returns the value of the `Number` object raised to the power of the value provided by the parameter specified.

Syntax

```
const Number power(
   const Number &val) const;
```

Parameter	Description
`val`	The power to which the number has to be raised.

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.

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.

setNull()

This method sets the value of the `Number` object to `NULL`.

Syntax

```
void setNull();
```

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.

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 is equal to 0, then create a Date object using integer parameters is returned. If the Number object is positive, then 1 is returned.

Syntax

```
const int sign() const;
```

sin()

This method returns the sin of the Number object.

Syntax

```
const Number sin();
```

squareroot()

This method returns the square root of the Number object.

Syntax

```
const Number squareroot() const;
```

tan()

This method returns the tangent of the `Number` object.

Syntax

```
const Number tan() const;
```

toBytes()

This method converts the `Number` object into a `Bytes` object. The bytes representation is assumed to be in length excluded format, that is, the `Byte.length()` method gives the length of valid bytes and the `0`th byte is the exponent byte.

Syntax

```
Bytes toBytes() const;
```

toText()

Convert the `Number` object to a formatted `string` or `UString` based on the parameters specified.

See Also: *Oracle Database SQL 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.

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.

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

This class enables the type definer to specify when a class is capable of having persistent or transient instances. Instances of classes derived from `PObject` are either persistent or transient. A class (called "`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 of the methods provided, such as lock(), apply only for persistent instances, not for transient instances.

Table 10–29 Summary of PObject Methods

Method	Summary
PObject() on page 10-195	`PObject` class constructor.
flush() on page 10-195	Flushes a modified persistent object to the database server.
getConnection() on page 10-196	Return the connection from which the `PObject` object was instantiated.
getRef() on page 10-163	Return a reference to a given persistent object.
isLocked() on page 10-196	Test whether the persistent object is locked.
isNull() on page 10-196	Test whether the object is `NULL`.
lock() on page 10-196	Lock a persistent object on the database server. The default mode is to wait for the lock if not available.
markDelete() on page 10-197	Mark a persistent object as deleted.
markModified() on page 10-197	Mark a persistent object as modified or dirty.
operator=() `on page 10-197`	Assign one `PObject` to another.

Table 10–29 (Cont.) Summary of PObject Methods

Method	Summary
operator delete() on page 10-198	Remove the persistent object from the application cache only.
operator new() on page 10-198	Creates a new persistent / transient instance.
pin() on page 10-199	Pins an object.
setName() on page 10-33	Sets the object value to `NULL`.
unmark() on page 10-200	Unmarks an object as dirty.
unpin() on page 10-200	Unpins an object. In the default mode, the pin count of the object is decremented by one.

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.

flush()

This method flushes a modified persistent object to the database server.

Syntax

```
void flush();
```

getConnection()

Returns the connection from which the persistent object was instantiated.

Syntax

```
const Connection *getConnection() const;
```

getRef()

This method returns a reference to the persistent object.

Syntax

```
RefAny getRef() const;
```

isLocked()

This method test whether the persistent object is locked. If the persistent object is locked, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isLocked() const;
```

isNull()

This method tests whether the persistent object is `NULL`. If the persistent object is `NULL`, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

lock()

Locks a persistent object on the database server.

Syntax

```
void lock(
   PObject::LockOption lock_option);
```

Parameter	Description
`lock_option`	Locking options: ■ `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 will wait 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; will not wait if the object is currently locked by another user; equivalent to `SELECT FOR UPDATE WITH NOWAIT`

markDelete()

This method marks a persistent object as deleted.

Syntax

```
void markDelete();
```

markModified()

This method marks a persistent object as modified or dirty.

Syntax

```
void mark_Modified();
```

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);
```

Parameter	Description
obj	The object from which the assigned value is obtained.

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

operator new()

This method is used to create 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 defualt 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.

Syntax	Description

`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.

`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.

pin()

This method pins the object and increments the pin count by one. As long as the object is pinned, it will not be freed by the cache even if there are no references to this object instance.

Syntax

```
void pin();
```

setNull()

This method sets the object value to `NULL`.

Syntax

```
void setNull();
```

unmark()

This method unmarks a persistent object as modified or deleted.

Syntax

```
void unmark();
```

unpin()

This method unpins a persistent object. In the default mode, the pin count of the object is decremented by one. When this method is invoked with `OCCI_PINCOUNT_RESET`, the pin count of the object is reset. If the pin count is reset, this method invalidates all the references (`Ref`) 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 pin count should be decremented or reset to `0`. Valid values are: ■ `OCCI_PINCOUNT_RESET` ■ `OCCI_PINCOUNT_DECR`

Producer Class

The `Producer` enqueues `Messages` into a queue and defines the enqueue options.

Table 10–30 Constants of the Producer Class

Constant	Description
`ENQ_BEFORE`	The message is enqueued before the message specified by the relatie message id.
`ENQ_TOP`	The message is enqueued before any other messages.
`ENQ_ON_COMMIT`	Default. The enqueue is part of the current transaction. The operation is complete when the transaction commits.
`ENQ_IMMEDIATE`	The enqueue is not part of the current transaction. The operation constitutes a transaction of its own.

For typical usage, please see Example 10–5 and Example 10–6 starting on page 10-78.

Table 10–31 Summary of Producer Methods

Method	Summary
Producer() on page 10-202	`Producer` class constructor.
getQueueName() on page 10-202	Retrieves the name of a queue on which the `Messages` will be enqueued.
getRelativeMessageId() on page 10-203	Retrieves the `Message` id that is referenced in a sequence deviation operation.
getSequenceDeviation() on page 10-203	Retrieves information regarding whether the `Message` should be dequeued ahead of other `Messages` in the queue.
getTransformation() on page 10-203	Retrieves the transformation applied before a `Message` is enqueued.
getVisibility() on page 10-203	Retrieves the transactional behavior of the enqueue request.
isNull() on page 10-204	Tests whether the `Producer` is `NULL`.
send() `on page 10-204`	Enqueues and sends a `Message`.
setNull() on page 10-204	Frees memory if the scope of the `Producer` extends beyond the `Connection` on which it was created.

Table 10–31 (Cont.) Summary of Producer Methods

Method	Summary
setQueueName() on page 10-205	Specifies the name of a queue on which the `Messages` will be enqueued.
setRelativeMessageId() on page 10-205	Specifies the `Message` id to be referenced in the sequence deviation operation.
setSequenceDeviation() on page 10-205	Specifies whether `Message` should be dequeued before other `Messages` already in the queue.
setTransformation() on page 10-206	Specify transformation applied before enqueuing a `Message`.
setVisibility() on page 10-206	Specify transaction behavior of the enqueue request.

Producer()

`Producer` object constructor.

Syntax	Description

`Producer(`
`   Connection *conn);`
	Creates a `Producer` object with the specified `Connection` handle.

`Producer(`
`   Connection *conn,`
`   const string& queue);`
	Creates a `Producer` object with the specified `Connection` handle and queue name.

Parameter	Description
`conn`	The connection of the new `Producer` object.
`queue`	The queue that will be used by the new `Producer` object.

getQueueName()

Retrieves the name of a queue on which the `Messages` will be enqueued.

Syntax

```
string getQueueName();
```

getRelativeMessageId()

Retrieves the `Message` id that is referenced in a sequence deviation operation. Used only if a sequence deviation is specified; ignored otherwise.

Syntax

```
Bytes getRelativeMessageId() const;
```

getSequenceDeviation()

Retrieves information regarding whether the `Message` should be dequeued ahead of other `Messages` in the queue. Valid return values are defined as constants of the `Producer` class in Table 10–30 on page 10-201: `ENQ_BEFORE` and `ENQ_TOP`.

Syntax

```
Producer::EnqueueSequence getSequenceDeviation() const;
```

getTransformation()

Retrieves the transformation applied before a `Message` is enqueued.

Syntax

```
Producer::string getTransformation() const;
```

getVisibility()

Retrieves the transactional behavior of the enqueue request. Valid return values are defined as constants of the `Producer` class in Table 10–30 on page 10-201: `ENQ_ON_COMMIT` and `ENQ_IMMEDIATE`.

Syntax

```
Producer::Visibility getVisibility() const;
```

isNull()

Tests whether the `Producer` is `NULL`. If the `Producer` is `NULL`, then `TRUE` is returned; otherwise, `FALSE` is returned.

Syntax

```
bool isNull() const;
```

send()

Enqueues and sends a `Message`.

Syntax	Description

`Bytes send(`
`   Message& message);`
	Used when queueName has been previously set by the setQueueName() method.

`Bytes send(`
`   Message& message,`
`   string& queueName);`
	Enqueue the `Message` to the specified `queueName`.

Parameter	Description
`message`	The `Message` that will be enqueued.
`queueName`	The name of a valid queue in the database.

setNull()

Frees memory associated with the `Producer`. Unless working in inner scope, this call should be made before terminating the `Connection`.

Syntax

```
void setNull();
```

setQueueName()

Specifies the name of a queue on which the `Messages` will be enqueued. Typically used when enqueuing multiple messages to the same queue.

Syntax

```
void setQueueName(
   const string& queueName);
```

Parameter	Description
`queueName`	The name of a valid queue in the database, to which the `Messages` will be enqueued.

setRelativeMessageId()

Specifies the `Message` id to be referenced in the sequence deviation operation. If the sequence deviation is not specified, this parameter will be ignored. Can be set for each enqueuing of a `Message`.

Syntax

```
void setRelativeMessageId(
   const Bytes& id);
```

Parameter	Description
`id`	The id of the relative `Message`.

setSequenceDeviation()

Specifies whether `Message` being enqueued should be dequeued before other `Message`(s) already in the queue. Can be set for each enqueuing of a `Message`.

Syntax

```
void setSequenceDeviation(
   unsigned int seqDev);
```

Parameter	Description
`seqDev`	The sequence deviation being set. Valid values are: ■ `ENQ_BEFORE` -- ■ `ENQ_TOP` --

setTransformation()

Specify transformation applied before enqueuing the `Message`.

Syntax

```
void setTransformation(
   string &transFunction);
```

Parameter	Description
`transFunction`	SQL transformation function.

setVisibility()

Specify transaction behavior of the enqueue request. Can be set for each enqueuing of a `Message`.

Syntax

```
void setVisibility(
   Visibility vis);
```

Parameter	Description
`visibility`	Visibility option being set. Valid values are: ■ `ENQ_ON_COMMIT` -- ■ `ENQ_IMMEDIATE` --

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 in place 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 10–32 Summary of Ref Methods

Method	Summary
Ref() on page 10-208	`Ref` object constructor.
clear() on page 10-208	Clears the reference.
getConnection() on page 10-208	Returns the connection this ref was created from.
isClear() on page 10-209	Checks if the Ref is cleared.
isNull() on page 10-209	This method checks if the `Ref` is `NULL`.
markDelete() on page 10-209	Marks the referred object as deleted.
operator->() on page 10-209	De-reference the `Ref` and pins the object if necessary.
operator*() on page 10-210	This operator de-references the `Ref` and pins / fetches the object if necessary.
operator==() on page 10-210	Checks if the `Ref` and the pointer refer to the same object.
operator!=() on page 10-210	Checks if the `Ref` and the pointer refer to different objects.
operator=() on page 10-211	Assignment operator.
ptr() on page 10-211	De-references the `Ref` and pins / fetches the object if necessary.

Table 10–32 (Cont.) Summary of Ref Methods

Method	Summary
setPrefetch() on page 10-211	Specifies type and depth of the object attributes to be followed for prefetching.
setLock() on page 10-212	Sets the lock option for the object referred from this.
setNull() on page 10-213	Sets the Ref to `NULL`.
unmarkDelete() on page 10-213	Unmarks for delete the object referred by this.

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.

clear()

This method clears the `Ref` object.

Syntax

```
void clear();
```

getConnection()

Returns the connection from which the `Ref` object was instantiated.

Syntax

```
const Connection *getConnection() const;
```

isClear()

This method checks if `Ref` object is cleared.

Syntax

```
bool isClear();
```

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;
```

markDelete()

This method marks the referenced object as deleted.

Syntax

```
void markDelete();
```

operator->()

This method dereferences the `Ref` object and pins, or fetches the referenced object if necessary. This might result in prefetching a graph of objects if prefetch attributes of the referenced object are set.

Syntax	Description
`T *operator->();`	Dereferenes and pins or fetches a non-`const` `Ref` object.
`const T *operator->() const;`	Dereferences and pins or fetches a `const` `Ref` object.

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 need to be deleted. Destructor would be automatically called when it goes out of scope.

Syntax	Description
`T& operator*();`	Dereferenes and pins or fetches a non-`const` `Ref` object.
`const T& operator*() const;`	Dereferences and pins or fetches a `const` `Ref` object.

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;
```

Parameter	Description
`ref`	The Ref object of the object to be compared.

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.

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.

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.
`const T *ptr() const;`	Returns a pointer of a `const` `Ref` object.

setPrefetch()

Sets the prefetching options for the 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 `setPrefetch(type_name, depth)` should 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);`
	Get the value of a parameter as a `PObject`.

`void setPrefetch(`
`   unsigned int depth);`
	Get the value of a parameter as a `PObject`.

`void setPrefetch(`
`   const UString &typeName,`
`   unsigned int depth);`
	Get the value of a parameter as a `PObject`.

Parameter	Description
`typeName`	Type of the Ref attribute to be prefetched.
`depth`	Depth level to which the links should be followed.

setLock()

This method specifies how the object should be locked when dereferenced.

Syntax

```
void setLock(
   LockOptions);
```

Parameter	Description
`lockOptions`	The lock options. Valid values are: ■ `OCCI_LOCK_NONE` ■ `OCCI_LOCK_X` ■ `OCCI_LOCK_X_NOWAIT`

setNull()

This method sets the `Ref` object to `NULL`.

Syntax

```
void setNull();
```

unmarkDelete()

This method unmarks the referred object as dirty.

Syntax

```
void unmarkDelete();
```

RefAny Class

The `RefAny` class is designed to support a reference to any type. Its primary purpose is to handle generic references and allow conversions of `Ref` in the type hierarchy. A `RefAny` object can be used as an intermediary between any two types, `Ref<`*x*`>` and `Ref<`*y*`>`, where *x* and *y* are different types.

Table 10–33 Summary of RefAny Methods

Method	Summary
RefAny() `on page 10-214`	Constructor for `RefAny` class.
clear() `on page 10-215`	Clear the reference.
getConnection() `on page 10-215`	Return the connection this ref was created from.
getUString() `on page 10-215`	Returns `UString` object at the current position of the `RefAny` object; globalization enabled.
isNull() `on page 10-216`	Check if the `RefAny` object is `NULL`.
markDelete() `on page 10-216`	Mark the object as deleted.
operator=() `on page 10-216`	Assignment operator.
operator==() `on page 10-216`	Check if this `RefAny` object is equal to a specified `RefAny`.
operator!=() `on page 10-217`	Check if not equal.
setObject() `on page 10-217`	Set the value of a parameter using an object.
setUString() `on page 10-218`	Set a the given `UString` in the `RefAny` object.
unmarkDelete() `on page 10-218`	Unmark the object as deleted.

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

clear()

This method clears the reference.

Syntax

```
void clear();
```

getConnection()

Returns the connection from which this reference was instantiated.

Syntax

```
const Connection* getConnection() const;
```

getUString()

Returns `UString` object at the current position of the `RefAny` object; globalization enabled.

Syntax

```
UString getUString() const;
   const UString &str);
```

isNull()

Returns `TRUE` if the object pointed to by this ref is `NULL` else `FALSE`.

Syntax

```
bool isNull() const;
```

markDelete()

This method marks the referred object as deleted.

Syntax

```
void markDelete();
```

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

```
RefAny& operator=(
   const RefAny& src);
```

Parameter	Description
`src`	The source RefAny object to be assigned.

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.

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.

setObject()

Set 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 will be converted to the corresponding SQL type before being sent to the database.

Syntax	Description

`void setObject(`
`   cosnt PObject *objptr,`
`   const string &sqltyp);`
	Set the value of a parameter using an object.

`void setObject(`
`   PObject* objptr,`
`   const USting &sqltyp);`
	Set the value of a parameter using an object; globalization enabled.

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...
`val`	The object containing the input parameter value.
`sqltyp`	The SQL type name of the object to be set; should be in the character set associated with the connection.

setUString()

Set a the given `UString` in the `RefAny` object; globalization enabled.

Syntax

```
void setUString(
   const UString &str);
```

Parameter	Description
`str`	The value.

unmarkDelete()

This method unmarks the referred object as dirty.

Syntax

```
void unmarkDelete();
```

ResultSet Class

A `ResultSet` provides access to a table of data generated by executing a `Statement`. Table rows are retrieved in sequence. Within a row, column values can be accessed in any order.

A `ResultSet` maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The `next` method moves the cursor to the next row.

The `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.

```
enum Status
{
   END_OF_FETCH = 0,
   DATA_AVAILABLE,
   STREAM_DATA_AVAILABLE
};
```

Table 10–34 Summary of ResultSet Methods

Method	Description
cancel() on page 10-222	Cancel the `ResultSet`.
closeStream() on page 10-222	Close the specified `Stream`.
getBDouble() on page 10-223	Return the value of a column in the current row as a `BDouble.`
getBfile() on page 10-223	Return the value of a column in the current row as a `Bfile.`
getBFloat() on page 10-223	Return the value of a column in the current row as a `BFloat.`
getBlob() on page 10-224	Return the value of a column in the current row as a `Blob` object.

Table 10–34 (Cont.) Summary of ResultSet Methods

Method	Description
getBytes() on page 10-224	Return the value of a column in the current row as a `Bytes` array.
getCharSet() on page 10-224	Return the character set in which data would be fetched.
getCharSetUString() on page 10-225	Return the character set in which data would be fetched as a `UString`.
getClob() on page 10-225	Return the value of a column in the current row as a `Clob` object.
getColumnListMetaData() on page 10-225	Return the describe information of the result set columns as a `MetaData` object.
getCurrentStreamColumn() on page 10-226	Return the column index of the current readable `Stream`.
getCurrentStreamRow() on page 10-226	Return the current row of the `ResultSet` being processed.
getCursor() on page 10-226	Return the nested cursor as a `ResultSet`.
getDate() on page 10-227	Return the value of a column in the current row as a `Date` object.
getDatebaseNCHARParam() on page 10-227	Returns whether data is in NCHAR character set or not.
getBDouble() on page 10-228	Return an IEEE754 double value as a C++ double.
getDouble() on page 10-228	Return the value of a column in the current row as a C++ double.
getBFloat() on page 10-228	Return an IEEE754 float value as a C++ float.
getFloat() on page 10-229	Return the value of a column in the current row as a C++ float.
getInt() on page 10-229	Return the value of a column in the current row as a C++ int.
getIntervalDS() on page 10-229	Return the value of a column in the current row as a `IntervalDS`.
getIntervalYM() on page 10-230	Return the value of a column in the current row as a `IntervalYM`.
getMaxColumnSize() on page 10-230	Return the maximum amount of data to read from a column.

Table 10–34 (Cont.) Summary of ResultSet Methods

Method	Description
getNumArrayRows() on page 10-230	Return the actual number of rows fetched in the last array fetch when `next(int numRows)` returned `END_OF_DATA`.
getNumber() on page 10-231	Return the value of a column in the current row as a `Number` object.
getObject() on page 10-231	Return the value of a column in the current row as a `PObject`.
getRef() on page 10-231	Return the value of a column in the current row as a `Ref`.
getRowid() on page 10-232	Return the current `ROWID` for a `SELECT FOR UPDATE` statement.
getRowPosition() on page 10-232	Return the row id of the current row position.
getStatement() on page 10-232	Return the `Statement` of the `ResultSet`.
getStream() on page 10-233	Return the value of a column in the current row as a `Stream`.
getString() on page 10-233	Return the value of a column in the current row as a string.
getTimestamp() on page 10-233	Return the value of a column in the current row as a `Timestamp` object.
getUInt() `on page 10-234`	Return the value of a column in the current row as a C++ unsigned int
getUString() `on page 10-234`	Return the value of a column in the current row as a `UString`.
getVector() `on page 10-235`	Return the specified collection parameter as a vector.
getVectorOfRefs() on page 10-237	Returns the column in the current position as a vector of `Refs`.
isNull() on page 10-238	Check whether the value is `NULL`.
isTruncated() on page 10-238	Check whether truncation has occurred.
next() on page 10-239	Make the next row the current row in a `ResultSet`.
preTruncationLength() on page 10-239	Return the actual length of the parameter before truncation.
setBinaryStreamMode() on page 10-240	Specify that a column is to be returned as a binary stream.

Table 10–34 (Cont.) Summary of ResultSet Methods

Method	Description
setCharacterStreamMode() on page 10-240	Specify that a column is to be returned as a character stream.
setCharSet() on page 10-241	Specify the character set in which the data is to be returned.
setCharSetUString() on page 10-241	Specify the character set in which the data is to be returned.
setDatebaseNCHARParam() on page 10-241	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() on page 10-242	Specify the data buffer into which data is to be read.
setErrorOnNull() on page 10-243	Enable/disable exception when `NULL` value is read.
setErrorOnTruncate() on page 10-243	Enable/disable exception when truncation occurs.
setMaxColumnSize() on page 10-244	Specify the maximum amount of data to read from a column.
status() on page 10-244	Return the current status of the `ResultSet`.

cancel()

This method cancels the result set.

Syntax

```
void cancel();
```

closeStream()

This method closes the stream specified by the parameter `stream`.

Syntax

```
void closeStream(
   Stream *stream);
```

Parameter	Description
stream	The Stream to be closed.

getBDouble()

This method returns the value of a column in the current row as a `BDouble`. If the value is SQL `NULL`, the result is `NULL`.

Syntax

```
BDouble getBDouble(
   unsigned int colIndex) = 0;
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2,...

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,...

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) = 0;
```

Parameter	Description
`colIndex`	Column index, first column is `1`, second is `2`,...

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`,...

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`,...

getCharSet()

Gets the character set in which data would be fetched, as a string.

Syntax

```
string getCharSet(
   unsigned int paramIndex) const;
```

Parameter	Description
paramIndex	Parameter index, first parameter is 1, second is 2,...

getCharSetUString()

Gets the character set in which data would be fetched, as a string.

Syntax

```
UString getCharSetUString(
   unsigned int paramIndex) const;
```

Parameter	Description
paramIndex	Parameter index, first parameter is 1, second is 2,...

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,...

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 will return the value of the given column as a `PObject`. The type of the C++ object will be 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;
```

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` needs to 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;
```

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;
```

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 ename,
       CURSOR(SELECT  dname, loc FROM dept)
FROM emp WHERE ename = '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`,...

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`,...

getDatebaseNCHARParam()

Returns whether data is in `NCHAR` character set or not.

Syntax

```
bool getDatebaseNCHARParam(
   unsigned int paramIndex) const;
```

Parameter	Description
`paramIndex`	Parameter index, first parameter is `1`, second is `2`,...

getBDouble()

Gets the value of an IEEE754 `DOUBLE` column as a C++ `double`, which has been defined as an `OUT` bind. If the value is SQL `NULL`, the result is `0`.

Syntax

```
double getBDouble(
   unsigned int colIndex) const;
```

Parameter	Description
`colIndex`	Column index, first column is `1`, second is `2`,...

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`,...

getBFloat()

Gets the value of an IEEE754 `FLOAT` column as a C++ `float`, which has been defined as an `OUT` bind. If the value is SQL `NULL`, the result is `0`.

Syntax

```
float getBFloat(
   unsigned int colIndex) const;
```

Parameter	Description
colIndex	Column index, first column is 1, second is 2,...

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,...

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,...

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);
```

Parameter	Description
`colIndex`	Column index, first column is `1`, second is `2`,...

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`,...

getMaxColumnSize()

Get the maximum amount of data to read for a column.

Syntax

```
unsigned int getMaxColumnSize(
   unsigned int colIndex) const;
```

Parameter	Description
`colIndex`	Column index, first column is `1`, second is `2`,...

getNumArrayRows()

Returns the actual number of rows fetched in the last array fetch when next() returned `END_OF_DATA`. Returns the actual number of rows fetched in the final array fetch

Syntax

```
unsigned int getNumArrayRows() const;
```

getNumber()

Get the value of a column in the current row as a `Number` object. Returns the column value; if the value is SQL `NULL`, the result is `NULL`.

Syntax

```
Number getNumber(
   unsigned int colIndex);
```

Parameter	Description
`colIndex`	Column index, first column is `1`, second is `2`,...

getObject()

Returns a pointer to a `PObject` holding the parameter value.

Syntax	Description

`PObject * getObject(`
`   unsigned int paramIndex);`
	Get the value of a parameter as a `PObject`.

`PObject * getObject(`
`   unsigned int paramIndex`
`   const UString &sqltyp);`
	Get the value of a parameter as a `PObject`; globalization enabled.

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...
`sqltyp`	SQL Type name of the object.

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`,...

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`,...

getRowPosition()

Get the `rowid` of the current row position.

Syntax

```
Bytes getRowPosition() const;
```

getStatement()

This method returns the `statement` of the `ResultSet`.

Syntax

```
const Statement* getStatement() const;
```

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`,...

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`,...

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`,...

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`,...

getUString()

Return the value as a UString.

Note: 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,...

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.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<BFloat> &vect);`	Used for `BFloat` vectors.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<Blob> &vect);`	Used for `Blob` vectors.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<Clob> &vect);`	Used for `Clob` vectors.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<Date> &vect);`	Used for vectors of Date Class.
`void getVector( ResultSet *rs, unsigned int colIndex, vector<double> &vect);`	Used for vectors of `double` type.

Syntax	Description

`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.

`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<Ref> &vect);`
	Used for on vectors of Ref Class.

`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.

Syntax	Description

`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.

`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`,...
`vect`	The reference to the vector (`OUT` parameter).

getVectorOfRefs()

Returns the column in the current position as a vector of `REF`s. The column should be a collection type (varray or nested table) of `REF`s. 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`,...
`vect`	The reference to the vector of `REF`s (`OUT` parameter).

isNull()

A column may have the value of SQL `NULL`; `wasNull()` reports whether the last column read had this special value. Note that you must first call `get`*`xxx`*`()` on a column to try to read its value and then call `wasNull()` 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`,...

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;
```

Parameter	Description
`paramIndex`	Parameter index, first parameter is `1`, second is `2`,...

next()

A `ResultSet` is initially positioned before its first row; the first call to next makes the first row the current row; the second call makes the second row the current row, and so on. If a read-able stream from the previous row is open, it is implicitly closed. The `ResultSet's` warning chain is cleared when a new row is read.

For non-streamed mode, next() always returns `RESULT_SET_AVAILABLE` or `END_OF_DATA`. Data is available for `getxxx()` method when the `RESULT_SET_REMOVE_AVAILABLE` status is returned. When this version of next() is used, array fetches are done for data being fetched with the setDataBuffer() interface. This means that `getxxx()` methods should not be called. The `numRows` amount of data for each column would materialize in the buffers specified with the setDataBuffer() interface. With array fetches, stream input is allowed, so `getxxxStream()` methods can also be called, once for each column.

Returns one of following:

- `DATA_AVAILABLE` — call `getxxx()` or read data from buffers specified with setDataBuffer()
- `END_OF_FETCH` — no more data available. This is the last set of rows for array fetches. This value is defined to be `0`.
- `STREAM_DATA_AVAILABLE` — call the getCurrentStreamColumn() method and read stream

Syntax

```
Status next(
   unsigned int numRows =1);
```

Parameter	Description
`numRows`	Number of rows to fetch for array fetches.

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`,...

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);
```

Parameter	Description
`colIndex`	Column index, first column is `1`, second is `2`,...
`size`	The amount of data to be read as a binary stream.

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`,...
`size`	The amount of data to be read as a character stream.

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`,...
`charSet`	Desired character set, as a string.

setCharSetUString()

Specify the character set value as a `UString` in which the data is returned.

Syntax

```
UString setCharSetUString(
   unsigned int colIndex,
   string charSet);
```

Parameter	Description
`colIndex`	Column index, first column is `1`, second is `2`,...
`charSet`	Desired character set, as a string.

setDatebaseNCHARParam()

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`,...
`isNCHAR`	`TRUE` or `FALSE`.

setDataBuffer()

Specify a data buffer where data would be fetched. The *`buffer`* parameter is a pointer to a user allocated data buffer. The current length of data must be specified in the `length` parameter. The amount of data should not exceed the *`size`* parameter. Finally, `type` is the data type of the data. Only non OCCI and non C++ specific types can be used, such as STL string. OCCI classes like `Bytes` and `Date` cannot be used.

If setDataBuffer() is used to fetch data for array fetches, it should be called only once for each result set. Data for each row is assumed to be at `buffer (i-1)` location, where `i` is the row number. Similarly, the length of the data would be assumed to be at `(length+(i-1))`.

Syntax

```
void setDataBuffer(
   unsigned int colIndex,
   void *buffer,
   Type type,
   sb4 size = 0,
   ub2 *length = NULL,
   sb2 *ind = NULL,
   ub2 *rc = NULL);
```

Parameter	Description
`colIndex`	Column index, first column is `1`, second is `2`,...
`buffer`	Pointer to user-allocated buffer; if array fetches are done, it should have `numRows` * size bytes in it.
`type`	Type of the data that is provided (or retrieved) in the buffer.

Parameter	Description
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).

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,...
causeException	Enable exceptions if TRUE. Disable if FALSE.

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,...

Parameter	Description
`causeException`	Enable exceptions if `TRUE`. Disable if `FALSE`.

setMaxColumnSize()

Set the maximum amount of data to read for a column.

Syntax

```
void setMaxColumnSize(
   unsigned int colIndex,
   unsigned int max);
```

Parameter	Description
`colIndex`	Column index, first column is `1`, second is `2`,...
`max`	The maximum amount of data to be read.

status()

Returns the current status of the result set. The status method can be called repeatedly to find out the status of the result. Data is available for `get`*`xxx`* method when the `RESULT_SET_AVAILABLE` status is returned. Returns one of following:

- `DATA_AVAILABLE` — call `get`*`xxx`*`()` or read data from buffers specified with the `setDataBuffer` method
- `STREAM_DATA_AVAILABLE` — call `getCurrentStream()` and read stream
- `END_OF_FETCH`

Syntax

```
Status status() const;
```

SQLException Class

The `SQLException` class provides information on generated errors, their codes and associated messages.

Table 10–35 Summary of SQLException

Method	Description
SQLException() on page 10-245	`SQLException` constructor.
getErrorCode() on page 10-246	Return the database error code.
getMessage() on page 10-246	Return the error message `string` for this exception.
getNLSMessage() on page 10-246	Return the error message `string` for this exception (Unicode support).
getNLSUStringMessage() on page 10-246	Return the error message `UString` for this exception (Unicode)support.
getUStringMessage() on page 10-247	Return the error message `UString` for this exception.
getXAErrorCode() on page 10-246	Return the error message string for this exception.
setErrorCtx() on page 10-248	Set the error context.

SQLException()

This is the `SQLException` constructor.

Syntax	Description
`SQLException();`	Constructs a `NULL SQLException` object.

`SQLException(`
`   const SQLException &src);`
	Constructs an `SQLException` object as a copy of another `SQLException` object.

Parameter	Description
`src`	The `SQLException` to be copied.

getErrorCode()

Gets the database error code.

Syntax

```
int getErrorCode() const;
```

getMessage()

Returns the error message string of this `SQLException` if it was created with an error message string. Returns `NULL` if the `SQLException` was created with no error message.

Syntax

```
string getMessage() const;
```

getNLSMessage()

Returns the error message `string` of this `SQLException` if it was created with an error message `string`. Passes the globalization enabled environment. Returns a `NULL string` if the `SQLException` was created with no error message. The error message will be in the character set associated with the environment.

Syntax

```
UString getNLSMessage(
   Environment *env) const;
```

Parameter	Description
`env`	The globalization enabled environment.

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 will be in the character set associated with the environment.

Syntax

```
UString getNLSUStringMessage(
   Environment *env) const;
```

Parameter	Description
env	The globalization enabled environment.

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 will be in the character set associated with the environment.

Syntax

```
UString getUStringMessage() const;
```

getXAErrorCode()

Determine if the thrown exception is due to an XA or an SQL error.

Used by C++ XA applications with dynamic registration. Returns an XA error code if the exception is due to XA, or XA_OK otherwise.

Syntax

```
int getrXAErrorCode(
   const string &dbname);
```

Parameter	Description
dbname	The database name; same as the optional dbname provided in the Open String (and used when connecting to the Resource Manager).

setErrorCtx()

Sets the pointer to the error context.

Syntax

```
void setErrorCtx(
   void *ctx);
```

Parameter	Description
`ctx`	The pointer to the error context.

StatelessConnectionPool Class

This class represents a pool of stateless, authenticated connections to the database.

Example 10–8 Using a StatelessConnectionPool

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 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 connection:

- `openConn = 5`
- `busyConn = 5`

When the user wants another connection, the pool will open 2 new connections and return 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 by the .

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

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

Table 10–36 Summary of StatelessConnectionPool Methods

Method	Description
getAnyTaggedConnection() on page 10-251	Return a pointer to the connection object, without the restriction of a matching tag.
getAnyTaggedProxyConnection() on page 10-252	Return a proxy connection from a connection pool.
getBusyConnections() on page 10-253	Return the number of busy connections in the connection pool.
getBusyOption() on page 10-253	Return 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() on page 10-253	Return a pointer to the `Connection` object.
getIncrConnections() on page 10-254	Return the number of incremental connections in the connection pool.
getMaxConnections() on page 10-254	Return the maximum number of connections in the connection pool.
getMinConnections() on page 10-254	Return the minimum number of connections in the connection pool.
getOpenConnections() on page 10-254	Return the number of open connections in the connection pool.
getPoolName() on page 10-255	Return the name of the connection pool.
getProxyConnection() on page 10-255	Return a proxy connection from a connection pool.
getTimeOut() on page 10-256	Return the timeout period of a connection in the connection pool.
releaseConnection() on page 10-256	Release the connection back to the pool with an optional tag.
setBusyOption() on page 10-257	Specify 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() on page 10-257	Set the maximum, minimum, and incremental number of pooled connections for the connection pool.
setTimeOut() on page 10-258	Set the timeout period of a connection in the connection pool.

Table 10–36 (Cont.) Summary of StatelessConnectionPool Methods

Method	Description
terminateConnection() on page 10-259	Close the connection and remove it from the pool.

getAnyTaggedConnection()

Returns a pointer to the connection object, without the restriction of a matching tag.

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.

> **Note:** A `getTag()` call to the `Connection` verifies the connection tag received.

Syntax	Description

`Connection *getAnyTaggedConnection(`
`   string tag="");`
	Returns a pointer to the connection object from a homogeneous stateless connection pool, without the restriction of a matching tag

`Connection *getAnyTaggedConnection(`
`   const string &name,`
`   const string &password,`
`   const string tag="");`
	Returns a pointer to the connection object from a heterogeneous stateless connection pool, without the restriction of a matching tag

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.

getAnyTaggedProxyConnection()

Return a proxy connection from a connection pool.

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.

Note: A `getTag()` call to the connection verifies the connection tag received.

Syntax	Description

`Connection *getAnyTaggedProxyConnection(`
`   const string &name,`
`   string roles[],`
`   unsigned int numRoles,`
`   string tag="",`
`   Connection::ProxyType proxyType=Connection::PROXY_DEFAULT);`
	Get a proxy connection with role specifications from a connection pool.

`Connection *getAnyTaggedProxyConnection(`
`   const string &name,`
`   string tag="",`
`   Connection::ProxyType proxyType=Connection::PROXY_DEFAULT);`
	Get a proxy connection within role specifications from the connection pool.

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. `PROXY_DEFAULT` represents a database username

getBusyConnections()

Return the number of busy connections in the connection pool.

Syntax

```
unsigned int getBusyConnections() const;
```

getBusyOption()

Return the behavior of the stateless connection pool when:

- all the connections in the pool are busy, and
- the number of connections have reached maximum.

The possible return values are:

- `WAIT` means that the thread waits and blocks until the connection becomes free
- `NOWAIT` throws an error
- `FORCEGET` creates a new connection, although maximum number of connections is opened and all are busy

Syntax

```
BusyOption getBusyOption();
```

getConnection()

Return a pointer to the connection object for a either a stateless or statefull Connection Pool.

Syntax	Description

`Connection *getConnection(`
`   string tag="");`
	Returns an authenticated connection, with a connection pool username and password.

`Connection *getConnection(`
`   const string &name,`
`   const string &password,`
`   const string tag="");`
	Returns a pointer to the connection object from a heterogeneous stateless connection pool.

Parameter	Description
`username`	The database username.
`password`	The database password.
`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.

getIncrConnections()

Returns the number of incremental connections in the connection pool. This call is useful only in cases of homogeneous connection pools.

Syntax

```
unsigned int getIncrConnections() const;
```

getMaxConnections()

Returns the maximum number of connections in the connection pool.

Syntax

```
unsigned int getMaxConnections() const;
```

getMinConnections()

Returns the minimum number of connections in the connection pool.

Syntax

```
unsigned int getMinConnections() const;
```

getOpenConnections()

Returns the number of open connections in the connection pool.

Syntax

```
unsigned int getOpenConnections() const;
```

getPoolName()

Returns the name of the connection pool.

Syntax

```
string getPoolName() const;
```

getProxyConnection()

Returns a proxy connection from a connection pool.

Syntax	Description
`Connection *getProxyConnection( const string &name, string roles[], unsigned int numRoles, string tag="", Connection::ProxyType proxyType=Connection::PROXY_DEFAULT);`	Get a proxy connection with role specifications from a connection pool.
`Connection *getProxyConnection( const string &name, string tag="", Connection::ProxyType proxyType=Connection::PROXY_DEFAULT);`	Get a proxy connection without role specifications from a connection pool

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`	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.

Parameter	Description
`proxyType`	The type of proxy authentication to perform. `PROXY_DEFAULT` represents a database username

getStmtCacheSize()

Retrieves the size of the statement cache.

Syntax

```
unsigned int getStmtCacheSize() const;
```

getTimeOut()

Returns the timeout period of a connection in the connection pool.

Syntax

```
unsigned int getTimeOut() const;
```

releaseConnection()

Releases the connection back to the pool with an optional tag.

Syntax

```
void releaseConnection(
   Connection *conn,
   const string tag="");
```

Parameter	Description
`conn`	The connection to be released.
`tag`	The user defined tag associated with the connection. The default of this parameter is "", which untags the connection.

setBusyOption()

Specify the behavior of the stateless connection pool when:

- all the connections in the pool are busy, and
- the number of connections have reached maximum.

Syntax

```
void setBusyOption(
   BusyOption busyOption);
```

Parameter	Description
`busyOption`	Valid values: ■ `WAIT` means that the thread waits and blocks until the connection is free ■ `NOWAIT` throws an error ■ `FORCEGET` creates a new connection, although maximum number of connections is opened and all are busy.

Caution: When `busyOption` is set to `FORCEGET`, an attempt can be made to create more connections than the number that can be supported. In such cases, a request for new connections will return an error 'ORA 00018 -- Maximum number of sessions exceeded' that will be propagated to the pool user.

setPoolSize()

Set 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);
```

Parameter	Description
`maxConn`	The maximum number of connections in the connection pool.
`minConn`	The minimum number of connections in the connection pool. Considered only in homogeneous pools.
`incrConn`	The incremental number of connections for the connection pool. Considered only in homogeneous pools.

setTimeOut()

Set the timeout period of a connection in the connection pool. OCCI will terminate any connections related to this connection pool that have been idle for longe4r than the timeout period specified.

Syntax

```
void setTimeOut(
   unsigned int connTimeOut=0);
```

Parameter	Description
`connTimeOut`	The timeout period, given in seconds.

setStmtCacheSize()

Enables or disables statement caching.

- A nonzero value will enable statement caching, with a cache of specified size.
- A zero value will disable caching.

Note: 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 size);
```

Parameter	Description
`size`	The size of the statement cache

terminateConnection()

Close the connection and remove it from the pool.

Syntax

```
void terminateConnection(
   Connection *conn);
```

Parameter	Description
`conn`	The connection to be terminated

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. A DML returning insert / update / delete statement or stored procedure call may have `OUT` bind parameters. Finally, a stored procedure call statement may have 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 and DML returning statements with `OUT` bind parameters.

Table 10–37 Constants of the Statement Class

Constant	Description
`NEEDS_STREAM_DATA`	Output `Streams` must be written for the streamed `IN` bind parameters. If there is more than one streamed parameter, call the getCurrentStreamParam() method to find out the bind parameter needing the stream. If the statement is executed iteratively, call getCurrentIteration() to find the iteration for which stream needs to be written.
`PREPARED`	The `Statement` object is set to a query.
`RESULT_SET_AVAILABLE`	The getResultSet() method must be called to get the result set.
`STREAM_DATA_AVAILABLE`	Input `Streams` must be read for the streamed `OUT` bind parameters. If there is more than one streamed parameter, call the getCurrentStreamParam() method to find out the bind parameter needing the stream. If the statement is executed iteratively, call getCurrentIteration() to find the iteration for which stream needs to be read
`UNPREPARED`	The `Statement` object is not set to a query.

Table 10–37 (Cont.) Constants of the Statement Class

Constant	Description
`UPDATE_COUNT_AVAILABLE`	The getUpdateCount() method must be called to find out the update count

Table 10–38 Summary of Statement Methods

Method	Description
addIteration() on page 10-265	Add an iteration for execution.
closeResultSet() on page 10-265	Immediately releases a result set's database and OCCI resources instead of waiting for automatic release.
closeStream() on page 10-266	Close the stream specified by the parameter *`stream`*.
disableCaching() on page 10-266	Disables statement caching.
execute() on page 10-266	Execute the SQL statement.
executeArrayUpdate() on page 10-267	Execute insert/update/delete statements which use only the `setDataBuffer()` or stream interface for bind parameters.
executeQuery() on page 10-268	Execute a SQL statement that returns a single `ResultSet`.
executeUpdate() on page 10-268	Execute a SQL statement that does not return a `ResultSet`.
getAutoCommit() on page 10-269	Return the current auto-commit state.
getBfile() on page 10-269	Return the value of a `BFILE` as a `Bfile` object.
getBlob() on page 10-269	Return the value of a `BLOB` as a `Blob` object.
getBytes() on page 10-270	Return the value of a `SQL BINARY` or `VARBINARY` parameter as `Bytes`.
getCharSet() on page 10-270	Return the character set that is in effect for the specified parameter.
getClob() on page 10-270	Return the value of a `CLOB` as a `Clob` object.
getConnection() on page 10-271	Returns the connection from which the `Statement` object was instantiated.
getCurrentIteration() on page 10-271	Return the iteration number of the current iteration that is being processed.
getCurrentStreamIteration() on page 10-271	Return the current iteration for which stream data is to be read or written.

Table 10–38 (Cont.) Summary of Statement Methods

Method	Description
getCurrentStreamParam() on page 10-271	Return the parameter index of the current output Stream that must be read or written.
getCursor() on page 10-272	Return the `REF CURSOR` value of an OUT parameter as a `ResultSet`.
getDatabaseNCHARParam() on page 10-272	Return whether data is in NCHAR character set.
getDate() `on page 10-272`	Return the value of a parameter as a `Date` object
getBDouble() on page 10-273	Return the value of a parameter as an IEEE754 double.
getDouble() on page 10-273	Return the value of a parameter as a C++ double.
getBFloat() `on page 10-274`	Return the value of a parameter as an IEEE754 float.
getFloat() on page 10-274	Return the value of a parameter as a C++ float.
getInt() on page 10-274	Return the value of a parameter as a C++ int.
getIntervalDS() `on page 10-275`	Return the value of a parameter as a `IntervalDS` object.
getIntervalYM() on page 10-275	Return the value of a parameter as a `IntervalYM` object.
getMaxIterations() `on page 10-275`	Return the current limit on maximum number of iterations.
getMaxParamSize() on page 10-276	Return the current max parameter size limit.
getNumber() on page 10-276	Return the value of a parameter as a `Number` object.
getObject() on page 10-276	Return the value of a parameter as a `PObject`.
getOCIStatement() on page 10-277	Return the OCI statement handle associated with the `Statement`.
getRef() on page 10-277	Return the value of a REF parameter as `RefAny`
getResultSet() on page 10-277	Return the current result as a `ResultSet`.
getRowid() on page 10-278	Return the row id parameter value as a `Bytes` object.
getSQL() on page 10-278	Return the current SQL string associated with the `Statement` object.
getSQLUString() on page 10-278	Return the current SQL string associated with the `Statement` object; globalization enabled.

Table 10–38 (Cont.) Summary of Statement Methods

Method	Description
getStream() on page 10-278	Return the value of the parameter as a stream.
getString() on page 10-279	Return the value of the parameter as a string.
getTimestamp() on page 10-279	Return the value of the parameter as a `Timestamp` object
getUInt() `on page 10-279`	Return the value of the parameter as a C++ unsigned int
getUpdateCount() on page 10-280	Return the current result as an update count for non-query statements.
getUString() on page 10-280	Return the value of a `UString`.
getVector() `on page 10-280`	Return the specified parameter as a vector.
getVectorOfRefs() `on page 10-283`	Returns the column in the current position as a vector of `REF`s.
isNull() on page 10-284	Check whether the parameter is `NULL`.
isTruncated() `on page 10-284`	Check whether the value is truncated.
preTruncationLength() `on page 10-284`	Returns the actual length of the parameter before truncation.
registerOutParam() on page 10-285	Register the type and max size of the OUT parameter.
setAutoCommit() on page 10-286	Specify auto commit mode.
setBfile() on page 10-286	Set a parameter to a `Bfile` value.
setBinaryStreamMode() on page 10-287	Specify that a column is to be returned as a binary stream.
setBlob() on page 10-287	Set a parameter to a `Blob` value.
setBytes() on page 10-288	Set a parameter to a `Bytes` array.
setCharacterStreamMode() on page 10-288	Specify that a column is to be returned as a character stream.
setCharSet() on page 10-288	Specify the character set for the specified parameter.
setClob() on page 10-289	Set a parameter to a `Clob` value.
setDate() on page 10-289	Set a parameter to a `Date` value.
setDatabaseNCHARParam() on page 10-290	Set to true if the data is to be in the NCHAR character set of the database; set to false to restore the default.

Table 10–38 (Cont.) Summary of Statement Methods

Method	Description
setDataBuffer() on page 10-290	Specify a data buffer where data would be available for reading or writing.
setDataBufferArray() on page 10-291	Specify an array of data buffers where data would be available for reading or writing.
setBDouble() on page 10-293	Set a parameter to an IEEE double value.
setDouble() on page 10-294	Set a parameter to a C++ double value.
setErrorOnNull() on page 10-294	Enable/disable exceptions for reading of `NULL` values.
setErrorOnTruncate() on page 10-295	Enable/disable exception when truncation occurs.
setBFloat() on page 10-293	Set a parameter to an IEEE float value.
setFloat() on page 10-295	Set a parameter to a C++ float value.
setInt() on page 10-295	Set a parameter to a C++ int value.
setIntervalDS() on page 10-296	Set a parameter to a `IntervalDS` value.
setIntervalYM() on page 10-296	Set a parameter to a `IntervalYM` value.
setMaxIterations() on page 10-297	Set the maximum number of invocations that will be made for the DML statement.
setMaxParamSize() on page 10-297	Set the maximum amount of data that can sent or returned from the parameter.
setNull() on page 10-298	Set a parameter to SQL `NULL`.
setNumber() on page 10-298	Set a parameter to a `Number` value.
setObject() on page 10-298	Set the value of a parameter using an object.
setPrefetchMemorySize() on page 10-299	Set the amount of memory that will be used internally by OCCI to store data fetched during each round trip to the server.
setPrefetchRowCount() on page 10-299	Set the number of rows that will be fetched internally by OCCI during each round trip to the server.
setRowid() on page 10-300	Set a row id bytes array for a bind position.
setSQL() `on page 10-300`	Associate new SQL string with `Statement` object.
setSQLUString() on page 10-301	Associate new SQL string with `Statement` object; globalization enabled.
setString() `on page 10-301`	Set a parameter for a specified index.

Table 10–38 (Cont.) Summary of Statement Methods

Method	Description
setTimestamp() on page 10-301	Set a parameter to a `Timestamp` value.
setUInt() on page 10-302	Set a parameter to a C++ unsigned int value.
setUString() `on page 10-302`	Set a parameter for a specified index; globalization enabled.
setVector() on page 10-303	Set a parameter to a vector of unsigned int.
setVectorOfRefs() on page 10-313	Sets a parameter to a vector; should be used when the type is a collection of `REF`s.
status() on page 10-314	Return the current status of the statement. This is useful when there is streamed data to be written.

addIteration()

After specifying set parameters, an iteration is added for execution.

Syntax

```
void addIteration();
```

closeResultSet()

In many cases, it is desirable to immediately release a result set's database and OCCI resources instead of waiting for this to happen when it is automatically closed; the closeResultSet() method provides this immediate release.

Syntax

```
void closeResultSet(
   ResultSet *resultSet);
```

Parameter	Description
`resultSet`	The resultset to be closed. The resultset should have been obtained by a call to the getResultSet() method on this statement.

closeStream()

Closes the stream specified by the parameter `stream`.

Syntax

```
void closeStream(
   Stream *stream);
```

Parameter	Description
`stream`	The stream to ne closed.

disableCaching()

Disables statement caching. Used if a user wishes to destroy a statement instead of caching it. Effective only if statement caching is enabled.

Syntax

```
void disableCaching();
```

execute()

Executes an SQL statement that may return either a result set or an update count. The statement may have read-able streams which may have to be written, in which case the results of the execution may not be readily available. The returned value is one of the following defined constants of the `Statement` class (see Table 10–37 on page 10-260): `PREPARED`, `UNPREPARED`, `RESULT_SET_AVAILABLE`, `UPDATE_COUNT_AVAILABLE`, `NEEDS_STREAM_DATA`, and `STREAM_DATA_AVAILABLE`.

If only one `OUT` value is returned for each invocation of the DML returning statement, iterative executes can be performed for DML returning statements. If output streams are used for `OUT` bind variables, they must be completely read in order. The getCurrentStreamParam() method would indicate which stream needs to 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.

executeArrayUpdate()

Executes insert/update/delete statements which use only the setDataBuffer() or stream interface for bind parameters. The bind parameters must be arrays of size arrayLength parameter. The statement may have read-able streams which may have to be written. The returned value is one of the following, and are defined as constants of the `Statement` class in Table 10–37 on page 10-260: `PREPARED`, `UNPREPARED`, `UPDATE_COUNT_AVAILABLE`, `NEEDS_STREAM_DATA`, and `STREAM_DATA_AVAILABLE`The returned value is one of the following:

If only one `OUT` value is returned for each invocation of the DML returning statement, array executes can be done for DML returning statements also. If output streams are used for `OUT` bind variables, they must be completely read in order. The getCurrentStreamParam() method would indicate which stream needs to be read. Similarly, getCurrentIteration() would indicate the iteration for which data is available.

Note: 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.

executeQuery()

Execute 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.

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.

Syntax	Description
`unsigned int executeUpdate( const string &sql="");`	Executes a non-query statement.
`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.

getAutoCommit()

Returns the current auto-commit state.

Syntax

```
bool getAutoCommit() const;
```

getBfile()

Returns the value of a `BFILE` parameter as a `Bfile` object.

Syntax

```
Bfile getBfile(
   unsigned int paramIndex);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...

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,...

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,...

getCharSet()

Returns the character set 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,...

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,...

getConnection()

Returns the connection from which the `Statement` object was instantiated.

Syntax

```
const Connection* getConnection() const;
```

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;
```

getCurrentStreamIteration()

Returns the current parameter stream for which data is available.

Syntax

```
unsigned int getCurrentStreamIteration() const;
```

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` needs to be written, or there are no output `Stream` parameters in the prepared statement, this method returns `0`.

Syntax

```
unsigned int getCurrentStreamParam() const;
```

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:** If there are multiple REF CURSORs being returned due to a batched call, data from each cursor must be completely fetched before retrieving the next REF CURSOR and starting fetch on it.

Syntax

```
ResultSet * getCursor(
   unsigned int paramIndex);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2,...

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,...

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;
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2,...

getBDouble()

Gets 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,...

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,...

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,...

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,...

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);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...

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,...

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,...

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;
```

getMaxParamSize()

The maxParamSize limit (in bytes) is the maximum amount of data sent or returned for any parameter value; it only applies to character and binary types. If the limit is exceeded, the excess data is silently discarded. Returns the current max parameter size limit.

Syntax

```
unsigned int getMaxParamSize(
   unsigned int paramIndex) const;
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2,...

getNumber()

Get 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,...

getObject()

Get 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.

Note: This method may be used to read database-specific, abstract data types.

Syntax	Description

`PObject * getObject(`
`   unsigned int paramIndex);`
	Get the value of a parameter as a `PObject`.

`PObject * getObject(`
`   unsigned int paramIndex`
`   const UString &sqltyp);`
	Get the value of a parameter as a `PObject`; globalization enabled.

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...
`sqltyp`	SQL Type name of the object.

getOCIStatement()

Get the OCI statement handle associated with the `Statement`.

Syntax

```
LNOCIStmt * getOCIStatement() const;
```

getRef()

Get the value of a `REF` parameter as `RefAny`. Returns the parameter value.

Syntax

```
RefAny getRef(
   unsigned int paramIndex);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...

getResultSet()

Returns the current result as a `ResultSet`.

Syntax

```
ResultSet * getResultSet();
```

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,...

getSQL()

Returns the current SQL string associated with the Statement object.

Syntax

```
string getSQL() const;
```

getSQLUString()

Returns the current SQL `UString` associated with the Statement object; globalization enabled.

Syntax

```
UString getSQLUString() const;
```

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,...

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,...

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,...

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,...

getUpdateCount()

Returns the current result as an update count.

Syntax

```
unsigned int getUpdateCount() const;
```

getUString()

Return the value as a UString.

Note: 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,...

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.

`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<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.`

Syntax	Description
`void getVector( Statement *stmt, unsigned int paramIndex, vector<float> &vect);`	Used for vectors of `float Class.`
`void getVector( Statement *stmt, unsigned int paramIndex, vector<int> &vect);`	Used for vectors of `int Class.`
`void getVector( Statement *stmt, unsigned int paramIndex, vector<IntervalDS> &vect);`	Used for vectors of IntervalDS Class.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<IntervalYM> &vect);`	Used for vectors of IntervalYM Class.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<Number> &vect);`	Used for vectors of Number Class.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<RefAny> &vect);`	Used for vectors of RefAny Class.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<string> &vect);`	Used for vectors of `string Class.`
`void getVector( Statement *stmt, unsigned int paramIndex, vector<T *> &vect);`	Intended for use on platforms where partial ordering of function templates is supported.
`void getVector( Statement *stmt, unsigned int paramIndex, vector<T> &vect);`	Intended for use on platforms where partial ordering of function templates is not supported, such as Windows NT. For `OUT` binds and DML returning statements.

Syntax	Description

`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.`

Parameter	Description
`stmt`	The statement.
`paramIndex`	Parameter index.
`vect`	Reference to the vector (`OUT` parameter) into which the values should be retrieved.

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 and DML returning clauses.

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,...

Parameter	Description
`vect`	The reference to the vector of `REF`s (`OUT` parameter). It is recommended to use `getVectorOfRefs()` instead of specialized `getVector()` function for `Ref<T>`.

isNull()

An `OUT` parameter may have the value of SQL `NULL`; `wasNull()` 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 `wasNull()` 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,...

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;
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...

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,...

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 `set`*`xxx`*`()` method for in/out parameters.

- When reading the value of an `out` parameter, you must use the `get`*`xxx`*`()` 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="");`
	Register the `type` of each `out` parameter of a PL/SQL stored procedure.

`void registerOutParam(`
`   unsigned int paramIndex,`
`   Type type,`
`   unsigned int maxSize=0,`
`   const UString &schName,`
`   const UString &sqltype);`
	Register the `type` of each `out` parameter of a PL/SQL stored procedure; globalization enabled.

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...
`type`	SQL type code defined by *`type`*; only datatypes corresponding to OCCI data types such as `Date`, `Bytes`, and so on.
`maxSize`	The maximum size of the retrieved value. For datatypes of `OCCIBYTES` and `OCCISTRING`, `maxSize` should be greater than `0`.
`schName`	The schema name.
`sqltype`	The name of the type in the data base (used for types which have been created with `CREATE TYPE`).

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.

setBfile()

Set 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,...

Parameter	Description
`val`	The parameter value.

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 when have PL/SQL `IN` or `IN/OUT` arguments in the bind position.

Parameter	Description
`colIndex`	Column index; first column is 1, second is 2,...
`size`	The amount of data to be read or returned as a binary `Stream`.
`inArg`	Pass `TRUE` if the bind position is a PL/SQL `IN` or `IN/OUT` argument

setBlob()

Set 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,...
`val`	The parameter value.

setBytes()

Set 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,...
`val`	The parameter value.

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.

`void setCharacterStreamMode(`
`   unsigned int colIndex,`
`   unsigned int size,`
`   bool inArg);`
	Sets column returned to be a character stream; used when have PL/SQL `IN` or `IN/OUT` arguments in the bind position.

Parameter	Description
`colIndex`	Column index; first column is 1, second is 2,...
`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

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,...
`charSet`	Selected character set, as a string.

setClob()

Set 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,...
`val`	The parameter value.

setDate()

Set 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,...
val	The parameter value.

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);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2,...
isNCHAR	TRUE if this parameter contains data in Database's NCHAR character set; FALSE otherwise

setDataBuffer()

Specifies a data buffer where data would be available. Also, used for OUT bind parameters of callable statements (and DML returning OUT binds in future).

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)`.

Syntax

```
void setDataBuffer(
   unsigned int paramIndex,
   void *buffer,
   Type type,
   sb4 size,
   ub2 *length,
   sb2 *ind = NULL,
   ub2 *rc= NULL);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...
`buffer`	Pointer to user-allocated buffer; if iterative or array executes are done, it should have `numIterations()` size bytes in it.
`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.
`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`	Return code; for iterative and array executes, a return code for every buffer element.

setDataBufferArray()

Specify 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 by the `arraySize` parameter. Note that 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.

Note: For `OUT` and `IN/OUT` binds, the number of elements returned in the array is returned in `arrayLength` as well. The client must make sure that it has allocated size `*arraySize` bytes for the `buffer`.

Syntax

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

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...

Parameter	Description
`buffer`	Pointer to user-allocated buffer. It should have `size* arraySize` bytes in it.
`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.
`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.

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) = 0;
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...
`dval`	The parameter value.

setBFloat()

Sets an IEEE754 float as a bind value to a `Statement` object at the position specified by the `paramIndex` attribute.

Syntax

```
void setBFloat(
   unsigned int paramIndex,
   const BFloat fval) = 0;
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2,...
fval	The parameter value.

setDouble()

Sets a parameter to a C++ double value.

Syntax

```
void setDouble(
   unsigned int paramIndex,
   double val);
```

Parameter	Description
paramIndex	Parameter index; first parameter is 1, second is 2,...
val	The parameter value.

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,...
causeException	Enable exceptions if `TRUE`. Disable if `FALSE`.

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,...
`causeException`	Enable exceptions if `TRUE`. Disable if `FALSE`.

setFloat()

Set a parameter to a C++ float value.

Syntax

```
void setFloat(
   unsigned int paramIndex,
   float val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...
`val`	The parameter value.

setInt()

Set 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,...
`val`	The parameter value.

setIntervalDS()

Set 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,...
`val`	The parameter value.

setIntervalYM()

Set a parameter to a `Interval` value.

Syntax

```
void setIntervalYM(
   unsigned int paramIndex,
   const IntervalYM &val);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...
`val`	The parameter value.

setMaxIterations()

Sets the maximum number of invocations that will be 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.

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,...
maxSize	The new maximum parameter size limit; must be >0.

setNull()

Set a parameter to SQL `NULL`. Note that you must specify the parameter's SQL type.

Syntax

```
void setNull(
   unsigned int paramIndex,
   Type type);
```

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...
`type`	SQL type.

setNumber()

Set 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,...
`val`	The parameter value.

setObject()

Set 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 will be converted to the corresponding SQL type before being sent to the database.

Syntax	Description

`void setObject(`
`   unsigned int paramIndex,`
`   PObject* val);`
	Set the value of a parameter using an object.

`void setObject(`
`   unsigned int paramIndex,`
`   PObject* val,`
`   const UString &sqltyp);`
	Set the value of a parameter using an object; globalization enabled.

Parameter	Description
`paramIndex`	Parameter index; first parameter is 1, second is 2,...
`val`	The object containing the input parameter value.
`sqltyp`	The SQL type name of the object to be set.

setPrefetchMemorySize()

Set the amount of memory that will be 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 to use for storing data fetched during each round trip to the server.

setPrefetchRowCount()

Set the number of rows that will be 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.

setRowid()

Set 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,...
`val`	The parameter value.

setSQL()

A new SQL string can be associated with a `Statement` object by 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, "", was used when the `Statement` was created, a `setSQL` method with the proper SQL string must be done prior to execution.

Syntax

```
void setSQL(
   const string &sql);
```

Parameter	Description
`sql`	Any SQL statement.

setSQLUString()

Associate an SQL statement with this object. Unicode support: the client `Environment` should be initialized in OCCIUTIF16 mode.

Syntax

```
void setSQLUString(
   const UString &sql);
```

Parameter	Description
`sql`	Any SQL statement, in character set associated with the connection source of the statement.

setString()

Set 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,...
`val`	The parameter value.

setTimestamp()

Set 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,...
val	The parameter value.

setUInt()

Set 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,...
val	The parameter value.

setUString()

Set 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,...
val	The parameter value.

setVector()

Sets a parameter to a vector. This method should be used 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 OCCI_STD_NAMESPACE::vector< T > &vect, const OCCI_STD_NAMESPACE::string &schemaName, const OCCI_STD_NAMESPACE::string &typeName);`	Intended for use on platforms where partial ordering of function templates is not supported, such as Windows NT. Multibyte support.
`void setVector( Statement *stmt, unsigned int paramIndex, const OCCI_STD_NAMESPACE::vector<T* > &vect, const OCCI_STD_NAMESPACE::string &schemaName, const OCCI_STD_NAMESPACE::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 std::vector<Bfile> &vect, const UString &sqltype;`	Sets a `const Bfile` vector
`void setVector( Statement *stmt, unsigned int paramIndex, const std::vector<Blob> &vect, const UString &sqltype;`	Sets a `const Blob` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, const std::vector<Clob> &vect, const UString &sqltype;`	Sets a `const Clob` vector.

Syntax	Description
`void setVector( Statement *stmt, unsigned int paramIndex, const std::vector<double> &vect, const UString &sqltype;`	Sets a `const double` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, const std::vector<float> &vect, const UString &sqltype;`	Sets a `const float` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, const std::vector<int> &vect, const UString &sqltype;`	Sets a `const int` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, const std::vector<IntervalDS> &vect, const UString &sqltype;`	Sets a `const IntervalDS` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, const std::vector<Number> &vect, const UString &sqltype;`	Sets a `const Number` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, const std::vector<RefAny> &vect, const UString &sqltype;`	Sets a `const RefAny` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, const std::vector<Timestamp> &vect, const UString &sqltype;`	Sets a `const Timestamp` vector.

Syntax	Description

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const std::vector<unsigned int> &vect,`
`   const UString &sqltype;`
	Sets a `const unsigned int` vector.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<BDouble> &vect`
`   const string &sqltype);`
	Sets a `BDouble` vector.

`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.

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<Date> &vect, const UString &sqltype;`	Sets a `const Date` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, const vector<double> &vect, const string &schemaName, const string &typeName);`	Sets a `const double` vector; multibyte support.

Syntax	Description

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<double> &vect,`
`   const UString &schemaName,`
`   const UString &typeName);`
	Sets a `const double` vector; UTF16 support.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<float> &vect,`
`   const string &schemaName,`
`   const string &typeName);`
	Sets a `const float` vector; multibyte support.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<float> &vect,`
`   const UString &schemaName,`
`   const UString &typeName);`
	Sets a `const float` vector; UTF16 support.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<int> &vect,`
`   const string &schemaName,`
`   const string &typeName);`
	Sets a `const int` vector; multibyte support.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<int> &vect,`
`   const UString &schemaName,`
`   const UString &typeName);`
	Sets a `const int` vector; UTF16 support.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<IntervalDS> &vect,`
`   const string &schemaName,`
`   const string &typeName);`
	Sets a `const IntervalDS` vector; multibyte support.

Syntax	Description

`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

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<IntervalYM> &vect,`
`   const UString &sqltype;`
	Sets a `const IntervalDS` vector.

`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.

Syntax	Description

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<RefAny> &vect,`
`   const string &schemaName,`
`   const string &typeName);`
	Sets a `const RefAny` vector; multibyte support.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<RefAny> &vect,`
`   const UString &schemaName,`
`   const UString &typeName);`
	Sets a `const RefAny` vector; UTF16 support.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<string> &vect,`
`   const string &schemaName,`
`   const string &typeName);`
	Sets a `const string` vector; multibyte support.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<string> &vect,`
`   const UString &schemaName,`
`   const UString &typeName);`
	Sets a `const string` vector; UTF16 support.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<Timestamp> &vect,`
`   const string &schemaName,`
`   const string &typeName);`
	Sets a `const Timestamp` vector; multibyte support.

`void setVector(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<Timestamp> &vect,`
`   const UString &schemaName,`
`   const UString &typeName);`
	Sets a `const Timestamp` vector; UTF16 support.

Syntax	Description

`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,`
`   const vector<UString> &vect,`
`   const UString &sqltype);`
	Sets a const `UString` vector; unicode support. The client `Environment` should be initialized in `OCCIUTIF16` mode.

`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.

Syntax	Description
`void setVector( Statement *stmt, unsigned int paramIndex, vector<double> &vect, string &sqltype;`	Sets a `double` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, vector<float> &vect, string &sqltype;`	Sets a `float` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, vector<int> &vect, string &sqltype;`	Sets an `int` vector .
`void setVector( Statement *stmt, unsigned int paramIndex, vector<IntervalDS> &vect, string &sqltype;`	Sets an `IntervalDS` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, vector<IntervalYM> &vect, string &sqltype;`	Sets an `IntervalYM` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, vector<Number> &vect, string &sqltype;`	Sets a `Number` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, vector<RefAny> &vect, string &sqltype;`	Sets a `RefAny` vector.

Syntax	Description
`void setVector( Statement *stmt, unsigned int paramIndex, vector<string> &vect, string &sqltype;`	Sets a `string` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, vector<Timestamp> &vect, string &sqltype;`	Sets a `Timestamp` vector.
`void setVector( Statement *stmt, unsigned int paramIndex, vector<unsigned int> &vect, string &sqltype;`	Sets an `unsigned int` vector.
`template <class T> void setVector( Statement *stmt, unsigned int paramIndex, const vector< T* > &vect, const string &sqltype);`	Intended for use on platforms where partial ordering of function templates is *not* supported.
`template <class T> void setVector( Statement *stmt, unsigned int paramIndex, const vector< T* > &vect, const UString &sqltype);`	Intended for use on platforms where partial ordering of function templates is not supported; globalization enabled.
`template <class T> void setVector( Statement *stmt, unsigned int paramIndex, const vector<Ref<T>> &vect, const UString &sqltype);`	Available only on platforms where partial ordering of function templates is supported. setVectorOfRefs() can be used instead; globalization enabled.
`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.

Syntax	Description

`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,...
`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

setVectorOfRefs()

Sets a parameter to a vector; should be used when the type is a collection of `REF`s are varrays or nested tables of `REF`s.

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; should be used when the type is a collection of `REF`s are varrays or nested tables of `REF`s.

`template  <class T>  void setVectorOfRefs(`
`   Statement *stmt,`
`   unsigned int paramIndex,`
`   const vector<Ref<T> > &vect,`
`   const string &sqltype) ;`
	Used for multibyte support.

Syntax	Description
`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.
`template <class T> void setVector( Statement *stmt, unsigned int paramIndex, const OCCI_STD_NAMESPACE::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,...
`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

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 needs to 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 is one of the following, and are defined as constants of the `Statement` class in Table 10–37 on page 10-260: `PREPARED`, `UNPREPARED`, `RESULT_SET_AVAILABLE`, `UPDATE_COUNT_AVAILABLE`, `NEEDS_STREAM_DATA`, and `STREAM_DATA_AVAILABLE`

```
Status status() const;
```

Stream Class

You use a `Stream` to read or write streamed data (usually `LONG`).

- A read-able `Stream` is used to obtain streamed data from a result set or `OUT` bind variable from a stored procedure call. A read-able `Stream` must be read completely until the end of data is reached or it should be closed to discard any unwanted data.
- A write-able `Stream` is used to provide streamed data (usually `LONG`) to parameterized statements including callable statements.

Table 10–39 Summary of Stream Methods

Method	Summary
readBuffer() on page 10-316	Reads the stream and returns the amount of data read from the Stream object.
readLastBuffer() on page 10-317	Reads last buffer from Stream.
writeBuffer() on page 10-317	Writes data from buffer to the stream.
writeLastBuffer() on page 10-317	Writes the last data from buffer to the stream.
status() on page 10-318	Returns the current status of the stream.

readBuffer()

Reads data from `Stream`. The size parameter specifies the maximum number of byte characters to read. Returns the amount of data read from the `Stream` object. -1 means end of data on the stream.

Syntax

```
int readBuffer(
   char *buffer,
   unsigned int size);
```

Parameter	Description
`buffer`	Pointer to data buffer; must be allocated and freed by caller.
`size`	Specifies the number of bytes to be read.

readLastBuffer()

This method 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

```
int readLastBuffer(
   char *buffer,
   unsigned int size);
```

Parameter	Description
`buffer`	Pointer to data buffer; must be allocated and freed by caller.
`size`	Specifies the number of bytes to be read.

writeBuffer()

Writes data from buffer to the stream. The amount of data written is determined by size.

Syntax

```
void writeBuffer(
   char *buffer,
   unsigned int size);
```

Parameter	Description
`buffer`	Pointer to data buffer.
`size`	Specifies the number of `chars` to be written.

writeLastBuffer()

This method writes the last data buffer to the stream. It can also be called to write the last chunk of data. The amount of data written is determined by size.

Syntax

```
void writeLastBuffer(
   char *buffer,
   unsigned int size);
```

Parameter	Description
`buffer`	Pointer to data buffer.
`size`	Specifies the number of bytes to be written.

status()

Returns the current status of the streams, which can be one of the following:

- `READY_FOR_READ`
- `READY_FOR_WRITE`
- `INACTIVE`

Syntax

```
Status status() const;
```

Subscription Class

The subscription class encapsulates the information and operations necessary for registering a subscriber for notification.

Table 10–40 Summary of Subscription Methods

Method	Summary
Subscription() on page 10-320	`Subscription` class constructor.
getCallbackContext() on page 10-320	Retrieves the callback context.
getDatabaseServersCount() on page 10-320	Retrieves the number of database servers in which the client is interested for the registration.
getDatabaseServerNames() on page 10-321	Returns the names of all the database servers where the client registered an interest for notification.
getPayload() on page 10-321	Retrieves the payload that has been set on the `Subscription` object prior to posting.
getSubscriptionName() on page 10-321	Retrieves the name of the `Subscription`.
getSubscriptionNamespace() on page 10-321	Retrieves the namespace of the `Subscription`.
getRecipientName() on page 10-322	Retrieves the name of the `Subscription` recipient.
getPresentation() on page 10-322	Retrieves the notification presentation mode.
getProtocol() on page 10-322	Retrieves the notification protocol.
setCallbackContext() on page 10-322	Registers a callback function for OCI protocol.
setDatabaseServerNames() on page 10-323	Specifies the database server distinguished names from which the client will receive notifications.
setNotifyCallback() on page 10-323	Specifies the context passed to user callbacks
setNull() on page 10-324	Specifies the `Subscription` object to `NULL` and frees the memory associated with the object.
setSubscriptionName() on page 10-325	Specifies the name of the subscription.

Table 10–40 (Cont.) Summary of Subscription Methods

Method	Summary
setSubscriptionNamespace() on page 10-326	Specifies the namespace in which the subscription is used.
setPayload() on page 10-324	Specifies the buffer content of the notification.
setRecipientName() on page 10-326	Specifies the name of the recipient of the notification.
setPresentation() on page 10-324	Specifies the presentation mode in which the client will receive notifications.
setProtocol() on page 10-325	Specifies the protocol in which the client will receive notifications.

Subscription()

`Subscription` class constructor.

Syntax

```
Subscription (
const Environment *env );
```

Parameter	Description
`env`	The `Environment`.

getCallbackContext()

Retrieves the callback context.

Syntax

```
void* getCallbackContext() const;
```

getDatabaseServersCount()

Returns the number of database servers in which the client is interested for the registration.

Syntax

```
unsigned int getDatabaseServersCount();
```

getDatabaseServerNames()

Returns the names of all the database servers where the client registered an interest for notification.

Syntax

```
vector<string> getDatabaseServerNames();
```

getPayload()

Retrieves the payload that has been set on the `Subscription` object prior to posting.

Syntax

```
void* getCPayload() const;
```

getSubscriptionName()

Retrieves the name of the subscription.

Syntax

```
const string& getSubscriptionName();
```

getSubscriptionNamespace()

Retrieves the namespace of the subscription. The subscription name must be consistent with its namespace; currently supports `Subscription::NS_AQ` and `Subscription::NS_Anonymous`.

Syntax

```
Namespace getSubscriptionNamespace();
```

getRecipientName()

Retrieves the name of the recipient of the notification. Possible return values are email address, the HTTP url and the PL/SQL procedure, depending on the protocol.

Syntax

```
const string& getRecipientName();
```

getPresentation()

Retrieves the presentation mode in which the client receives notifications. Valid values are `PRES_DEFAULT` and `PRES_XML`.

Syntax

```
unsigned int getPresentation();
```

getProtocol()

Retrieves the protocol in which the client receives notifications. Valid values are `PROTO_CBK`, `PROTO_MAIL`, `PROTO_HTTP`, and `PROTO_SERVER`.

Syntax

```
Protocol getProtocol();
```

setCallbackContext()

Registers a notification callback function when the protocol is set to `PROTO_CBK`. Context registration is also included in this call.

Syntax

```
void setCallbackContext(
   void *ctx);
```

Parameter	Description
ctx	The context set.

setDatabaseServerNames()

Specifies the list of database server distinguished names from which the client will receive notifications.

Syntax

```
void setDatabaseServerNames(
   const vector<string>& dbsrv);
```

Parameter	Description
dbsrv	The list of database distinguished names

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 needs to 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.

setNull()

Sets the Subscription object to NULL and frees the memory associated with the object.

Syntax

```
void setNull();
```

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.

setPresentation()

Set the presentation mode in which the client will receive notifications.

Syntax

```
void setPresentation(
   unsigned int presentation);
```

Parameter	Description
presentation	Presentation mode: ■ PRES_DEFAULT ■ PRES_XML

setProtocol()

Sets the protocol in which the client will receive event notifications:

- `PROTO_CBK` (default)
- `PROTO_MAIL` -- through e-mail
- `PROTO_SERVER` -- through an invoked PL/SQL procedeure in the database
- `PROTO_HTTP` -- through an HTTP URL

Syntax

```
void setProtocol(
   Protocol prot);
```

Parameter	Description
`prot`	Protocol mode

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.

setSubscriptionNamespace()

Sets the namespace in which the subscription is used. The subscription name must be consistent with its namespace. Default value is `NS_AQ`.

Syntax

```
void setSubscriptionNamespace(
   Namespace nameSpace);
```

Parameter	Description
`nameSpace`	Namespace in which the subscription handle is used. Valid values include `Subscription::NS_AQ` and `Subscription::NS_Anonymous`.

setRecipientName()

Sets the name of the recipient of the notification.

- For `PROTO_HTTP`, this is the HTTP URL to which the notification is sent, like `http://www.oracle.com:80`. The database does not check if the URL is valid.
- For `PROTO_MAIL`, this is the e-mail address to which the notification is sent, like `xyz@oracle.com`. The database does not check if the e-mail is valid.
- For `PROTO_SERVER`, this is the database procedure invoked by the notification, like `schema.procedure`. The subscriber must have the appropriate permissions on the procedure.

Syntax

```
void setRecipientName(
   const string& recipient);
```

Parameter	Description
`recipient`	Name of the notification recipient.

Timestamp Class

This class conforms to the SQL92 `TIMESTAMP` and `TIMESTAMPTZ` 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 fo 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: For GMT timezone, both hour and minute equal `0`.

This behaviour change also applies to setDate() and setTime() methods.

The fields of `Timestamp` class and their legal ranges are provided in Table 10–41. An `SQLException` will occur if a parameter is out of range.

Table 10–41 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`
`sec`	unsigned int	`0`	`61`
`tzhour`	int	`-12`	`14`

Table 10–41 (Cont.) Fields of Timestamp and Their Legal Ranges

Field	Type	Minimum Value	Maximum value
`tzmin`	int	`-59`	`59`

Table 10–42 Summary of Timestamp Methods

Method	Summary
Timestamp() on page 10-329	`Timestamp` class constructor.
fromText() on page 10-331	Sets the time stamp from the values provided by the string.
getDate() on page 10-333	Gets the date from the `Timestamp` object.
getTime() on page 10-333	Gets the time from the `TimeStamp` object.
getTimeZoneOffset() on page 10-334	Returns the time zone hour and minute offset value.
intervalAdd() on page 10-334	Returns a `Timestamp` object with value (this + interval).
intervalSub() on page 10-334	Returns a `Timestamp` object with value (this - interval).
isNull() on page 10-335	Check if `Timestamp` is `NULL`.
operator=() on page 10-335	Simple assignment.
operator==() on page 10-335	Check if a and b are equal.
operator!=() on page 10-336	Check if a and b are not equal.
operator>() on page 10-336	Check if a is greater than b.
operator>=() on page 10-337	Check if a is greater than or equal to b.
operator<() on page 10-337	Check if a is less than b.
operator<=() on page 10-338	Check if a is less than or equal to b.
setDate() on page 10-338	Sets the year, month, day components contained for this timestamp.
setNull() on page 10-338	Sets the value of Timestamp to `NULL`
setTime() on page 10-339	Sets the day, hour, minute, second and fractional second components for this timestamp.
setTimeZoneOffset() on page 10-339	Sets the hour and minute offset for time zone.

Table 10–42 (Cont.) Summary of Timestamp Methods

Method	Summary
subDS() on page 10-340	Returns a `IntervalDS` representing this - `val`.
subYM() on page 10-340	Returns a `IntervalYM` representing this - `val`.
toText() on page 10-340	Returns a `string` representation for the timestamp in the format specified.

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 OCCI_STD_NAMESPACE::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.

Syntax	Description

`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.

Example 10–9 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);
```

Example 10–10 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 10–11 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
}
```

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.

fromText()

Sets the timestamp value from the string. The string is expected to be in the format specified. If nlsParam is specified, this will determine 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.

Set `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 Reference* for information on `TO_DATE`

Syntax	Description

`void fromText(`
`   const string &timestampStr,`
`   const string &fmt = "",`
`   const string &nlsParam = "",`
`   const Environment *env = NULL);`
	Set `Timestamp` object to value represented by a `string`.

`void fromText(`
`   const UString &timestampStr,`
`   const UString &fmt,`
`   const UString &nlsParam,`
`   const Environment *env = NULL);`
	Set `Timestamp` object to value represented by a `UString`; globalization enabled.

Parameter	Description
`timestampStr`	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.

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.

getTime()

Returns the hour, minute, second, and fractional second components

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.

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.

intervalAdd()

Adds an interval to timestamp.

Syntax	Description

`Timestamp intervalAdd(`
`   const IntervalDS& val) const;`
	Adds an IntervalDS interval to the timestamp.

`Timestamp intervalAdd(`
`   const IntervalYM& val) const;`
	Adds an IntervalYM interval to the timestamp.

Parameter	Description
`val`	Interval to be added.

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

`Timestamp intervalSub(`
`   const IntervalDS& val) const;`
	Subtracts an IntervalDS interval to the timestamp.

`Timestamp intervalsUB(`
`   const IntervalYM& val) const;`
	Subtracts an IntervalYM interval to the timestamp.

Parameter	Description
`val`	Interval to be subtracted.

isNull()

Returns `TRUE` if `Timestamp` is `NULL` or `FALSE` otherwise.

Syntax

```
bool isNull() const;
```

operator=()

Assigns a given timestamp object to this object.

Syntax

```
Timestamp & operator=(
   const Timestamp &src);
```

Parameter	Description
`src`	Value to be assigned.

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);
```

Parameter	Description
first	First timestamp to be compared.
second	Second timestamp to be compared.

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.

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.

operator>=()

Compares the timestamps specified. If the `first` timestamp is greater than or equal to the `second` timestamp then `TRUE` is returned; otherwise, `FALSE` is returned. If either timestamp is `NULL` then `FALSE` is returned.

Syntax

```
bool operator>=(const Timestamp &first,
   const Timestamp &second);
```

Parameter	Description
`first`	First timestamp to be compared.
`second`	Second timestamp to be compared.

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.

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.

setDate()

Sets the year, month, day components contained for this timestamp

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.

setNull()

Set the timestamp to `NULL`.

Syntax

```
void setNull();
```

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.

setTimeZoneOffset()

Sets the hour and minute offset for time zone.

Syntax

```
void setTimeZoneOffset(
   int hour,
   int minute);
```

Parameter	Description
hour	Time zone hour. Valid values are -12 through 12.
minute	Time zone minute. Valid values are -59 through 59.

subDS()

Computes the difference between this timestamp and the specified timestamp and return the difference as an `IntervalDS`.

Syntax

```
IntervalDS subDS(
   const Timestamp& val) const;
```

Parameter	Description
val	Timestamp to be subtracted.

subYM()

Computes the difference between timestamp values and return the difference as an `IntervalYM`.

Syntax

```
IntervalYM subYM(
   const Timestamp& val) const;
```

Parameter	Description
`val`	`Timestamp` to be subtracted.

toText()

Return a `string` or `UString` representation for the timestamp in the format specified.

If `nlsParam` is specified, this will determine the nls parameters to be 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 Reference* for information on `TO_DATE`

Syntax	Description

`string toText(`
`   const string &fmt,`
`   unsigned int fsprec,`
`   const string &nlsParam = "") const;`
	Return a `string` representation for the timestamp in the format specified.

`UString toText(`
`   const UString &fmt,`
`   unsigned int fsprec,`
`   const UString &nlsParam) const;`
	Return 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`.

Index