Oracle® Database

JDBC Developer's Guide

12*c* Release 1 (12.1)

E49300-05

June 2014

This book describes how to use Oracle JDBC drivers to develop powerful Java database applications.

ORACLE®

Oracle Database JDBC Developer's Guide, 12*c* Release 1 (12.1)

E49300-05

Copyright © 1999, 2014, Oracle and/or its affiliates. All rights reserved.

Primary Author: Tulika Das, Venkatasubramaniam Iyer, Elizabeth Hanes Perry, Brian Wright, Thomas Pfaeffle

Contributing Author: Brian Martin

Contributor: The Oracle Database 12*c* documentation is dedicated to Mark Townsend, who was an inspiration to all who worked on this release.

Contributor: Kuassi Mensah, Douglas Surber, Paul Lo, Ed Shirk, Tong Zhou, Jean de Lavarene, Rajkumar Irudayaraj, Ashok Shivarudraiah, Angela Barone, Rosie Chen, Sunil Kunisetty, Joyce Yang, Mehul Bastawala, Luxi Chidambaran, Srinath Krishnaswamy, Swati Rao, Pankaj Chand, Aman Manglik, Longxing Deng, Magdi Morsi, Ron Peterson, Ekkehard Rohwedder, Catherine Wong, Scott Urman, Jerry Schwarz, Steve Ding, Soulaiman Htite, Anthony Lai, Prabha Krishna, Ellen Siegal, Susan Kraft, Sheryl Maring

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

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

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

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

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

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

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

4 Oracle Extensions

5 Features Specific to JDBC Thin

6 Features Specific to JDBC OCI Driver

7 Server-Side Internal Driver

Part III Connection and Security

8 Data Sources and URLs

9 JDBC Client-Side Security Features

10 Proxy Authentication

Part IV Data Access and Manipulation

11 Accessing and Manipulating Oracle Data

12 Java Streams in JDBC

13 Working with Oracle Object Types

14 Working with LOBs and BFILEs

15 Using Oracle Object References

16 Working with Oracle Collections

17 Result Set

18 JDBC RowSets

19 Globalization Support

Part V Performance and Scalability

20 Statement and Result Set Caching

21 Performance Extensions

22 OCI Connection Pooling

23 Database Resident Connection Pooling

24 Oracle Advanced Queuing

Part VIII Manageability

31 Database Administration

32 Diagnosability in JDBC

33 JDBC DMS Metrics

Part IX Appendixes

A JDBC Reference Information

List of Examples

List of Figures

List of Tables

Preface

This preface introduces you to the *Oracle Database JDBC Developer's Guide* discussing the intended audience, structure, and conventions of this document. A list of related Oracle documents is also provided.

This preface covers the following topics:

Audience

The *Oracle Database JDBC Developer's Guide* is intended for developers of Java Database Connectivity (JDBC)-based applications and applets. This book can be read by anyone with an interest in JDBC programming, but assumes at least some prior knowledge of the following:

- Java
- Oracle PL/SQL
- Oracle databases

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

The following books are also available from the Oracle Java Platform group:

- *Oracle Database Java Developer's Guide*

This book introduces the basic concepts of Java and provides general information about server-side configuration and functionality. Information that pertains to the Oracle Java platform as a whole, rather than to a particular product (such as JDBC) is in this book. This book also discusses Java stored procedures, which were formerly discussed in a standalone book.

- *Oracle Database SQLJ Developer's Guide*

  This book covers the use of SQLJ to embed static SQL operations directly into Java code, covering SQLJ language syntax and SQLJ translator options and features. Both standard SQLJ features and Oracle-specific SQLJ features are described.

- *Oracle Database JPublisher User's Guide*

  This book describes how to use the Oracle JPublisher utility to translate object types and other user-defined types to Java classes. If you are developing JDBC applications that use object types, VARRAY types, nested table types, or object reference types, then JPublisher can generate custom Java classes to map to them.

The following OC4J documents, for Oracle Application Server releases, are also available from the Oracle Java Platform group:

- *Oracle Application Server Containers for J2EE User's Guide*

  This book provides some overview and general information for OC4J; primer chapters for servlets, JSP pages, and EJBs; and general configuration and deployment instructions.

- *Oracle Containers for J2EE Support for JavaServer Pages Developer's Guide*

  This book provides information for JSP developers who want to run their pages in OC4J. It includes a general overview of JSP standards and programming considerations, as well as discussion of Oracle value-added features and steps for getting started in the OC4J environment.

- *Oracle Containers for J2EE JSP Tag Libraries and Utilities Reference*

  This book provides conceptual information and detailed syntax and usage information for tag libraries, JavaBeans, and other Java utilities provided with OC4J.

- *Oracle Containers for J2EE Servlet Developer's Guide*

  This book provides information for servlet developers regarding use of servlets and the servlet container in OC4J. It also documents relevant OC4J configuration files.

- *Oracle Containers for J2EE Services Guide*

  This book provides information about basic Java services supplied with OC4J, such as JTA, JNDI, and the Oracle Application Server Java Object Cache.

- *Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide*

  This book provides information about the EJB implementation and EJB container in OC4J.

The following documents are from the Oracle Server Technologies group:

- *Oracle Database Development Guide*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database PL/SQL Language Reference*
- *Oracle Database SQL Language Reference*

- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database Advanced Security Guide*
- *Oracle Database Reference*
- *Oracle Database Error Messages*

The following documents from the Oracle Application Server group may also be of some interest:

- *Oracle Application Server Administrator's Guide*
- *Oracle Fusion Middleware Administrator's Guide for Oracle HTTP Server*
- *Oracle Fusion Middleware Performance Guide*
- *Oracle Fusion Middleware Application Globalization Guide*
- *Oracle Fusion Middleware Administrator's Guide for Oracle Web Cache*
- *Oracle Fusion Middleware Upgrade Planning Guide*

The following are available from the JDeveloper group:

- Oracle JDeveloper online help
- Oracle JDeveloper documentation on the Oracle Technology Network:

  http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html

Printed documentation is available for sale in the Oracle Store at:

http://shop.oracle.com/

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

http://www.oracle.com/technetwork/community/join/why-join/index.html

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

http://www.oracle.com/technetwork/documentation/index.html

The following resources are available:

- Web site for JDBC, including the latest specifications:

  http://www.oracle.com/technetwork/java/javase/jdbc/index.htm

Conventions

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

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

Conventions in Text

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

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

Conventions in Code Examples

Code examples illustrate Java, SQL, and command-line statements. Examples 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.	SQL> SELECT NAME FROM V$DATAFILE; NAME ---------- /fsl/dbs/tbs_01.dbf /fs1/dbs/tbs_02.dbf . . . /fsl/dbs/tbs_09.dbf 9 rows selected.
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;

Conventions for Windows Operating Systems

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

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

Convention	Meaning	Example
File and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \\, then Windows assumes it uses the Universal Naming Convention.	`c:\winnt"\"system32 is the same as C:\WINNT\SYSTEM32`
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`
Special characters	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (') do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	`C:\>exp HR/hr TABLES=employees QUERY=\"WHERE job_id='SALESMAN' and salary<1600\" C:\>imp SYSTEM/`*`password`*` FROM USER=HR TABLES=(employees, dept)`
HOME_NAME	Represents the Oracle home name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	`C:\> net start Oracle`*`HOME_NAME`*`TNSListener`

Convention	Meaning	Example
ORACLE_HOME and *ORACLE_BASE*	In releases prior to Oracle8*i* release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *ORACLE_HOME* directory that by default used one of the following names: ■ `C:\orant` for Windows NT ■ `C:\orawin98` for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *ORACLE_HOME* directory. There is a top level directory called *ORACLE_BASE* that by default is `C:\oracle`. If you install the latest Oracle release on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is `C:\oracle\ora`*nn*, where *nn* is the latest release number. The Oracle home directory is located directly under *ORACLE_BASE*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle Database Platform Guide for Microsoft Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *ORACLE_BASE*`\`*ORACLE_HOME*`\rdbms\admin` directory.

Changes in This Release for Oracle Database JDBC Developer's Guide

This preface contains:

Changes in Oracle Database 12*c* Release 1 (12.1)

The following are changes in *Oracle Database JDBC Developer's Guide* for Oracle Database 12*c* Release 1 (12.1).

New Features

The following features are new in this release:

- Support for Row Count Per Iteration for Array DMLs

  See "Row Count per Iteration for Array DMLs" section on page 21-11.
- Support for Application Continuity

  See Chapter 27, "Application Continuity for Java".
- Support for Transaction Guard

  See Chapter 26, "Transaction Guard for Java".
- Database Resident Connection Pooling

  See Chapter 23, "Database Resident Connection Pooling".
- Support for Latest JDBC Standard

  See "Support for JDBC 4.1 Standard" section on page 3-11.
- Support for SHA-2 in Oracle Advanced Security

  See "Support for Oracle Advanced Security" section on page 9-1.
- Support for the `O7L_MR` Client Ability

  See "Support for Login Authentication" section on page 9-4.
- Support for Implicit Results

  See "Support for Implicit Results" section on page 2-18.
- Support for Invisible Columns

  See "Working with Invisible Columns" section on page 2-14.
- Support for PL/SQL Package Types as Parameters

See "Using PL/SQL Types" section on page A-6.

- Support for Monitoring of Database Operations

  See "Monitoring Database Operations" section on page 3-11.

- Support for Promoting a Local Transaction to a Global Transaction

  See "OracleXAResource Method Functionality and Input Parameters" on page 30-8.

- Support for the `?` character used in the `MATCH_RECOGNIZE` clause

  See "MATCH_RECOGNIZE Clause" section on page A-12.

- Support for specifying delay between connection retries

  See "Support for Delay in Connection Retries" section on page 8-11.

- Support for Increased Length Limit for Various Data Types

  Starting from this release, there is an increase in the length limit for VARCHAR2, NVARCHAR2, and RAW data types in Oracle SQL to 32767 (32K) bytes. For NVARCHAR columns, the limit is 32766 bytes.

  For using these data types with extended size, you must perform the following steps:

  1. Set the `COMPATIBLE` initialization parameter to 12.0.0.0.
  2. Set the `MAX_STRING_SIZE` initialization parameter to `EXTENDED`.
  3. Run the `rdbms/admin/utl32k.sql` script.

     See Also: *Oracle Database Reference* for more information about using these data types with extended size

Multitenant Data Source for Java

To enable sharing a common pool of connections across multiple PDBs (or tenants), Oracle JDBC and Universal Connection Pool (UCP) furnish the multitenant data source for Java. In Oracle Database 12*c* Release 1 (12.1), it is based on a combination of the following:

- Global database user (with access privileges to any PDB)
- UCP connection labeling
- The new `SET CONTAINER` statement within a callback function:

  The advantage of the `SET CONTAINER` statement is that the pool does not have to create a new connection to a PDB, if there is an existing connection to a different PDB. The pool can use the existing connection and can connect to the desired PDB through the `SET CONTAINER` statement. Use the following command to achieve this:

  ```
  ALTER SESSION SET CONTAINER=<PDB Name>
  ```

  This feature avoids the need to create a new connection from scratch.

- The UCP connection labeling callback interface.

Example

Following steps describe how mutlitenant data source for Java works:

1. Suppose, you have `Tenant1`, who calls the `getConnection` method with the corresponding label (mapped to the database ID) for a connection to `PDB1`.

2. UCP searches the pool for a free connection, tentatively with the specified label mapped to the database ID.
3. If for a connection `Conn1`, the label reads `PDB1`, then it is passed to JDBC, and consequently to `Tenant1` for using the connection.

   Otherwise, if no connection label reads `PDB1`, then the following steps are performed:

   a. UCP invokes the user-implemented `configure` callback method to set the connection to `PDB1` using the `ALTER SESSION SET CONTAINER` command.
   b. The `SET CONTAINER` statement is passed to the server and parsed.
   c. The server executes the `SET CONTAINER` statement and assigns the PDB-specific role to `Tenant1`[1].
   d. The server then connects `Conn1` to `PDB1`, and returns the corresponding database ID (`dbid`) and other properties to JDBC.
   e. JDBC notifies UCP and passes `Conn1` to `Tenant1` for using the connection.

   See Also:

   http://www.oracle.com/technetwork/database/application-development/index-099369.html

Deprecated Features

The following features are deprecated in this release, and may be desupported in a future release:

- Concrete classes in the `oracle.sql` package

  The concrete classes in the `oracle.sql` package are deprecated. Use the new JDBC interfaces instead of these classes.

  See MoS Note 1364193.1 for more information about these interfaces.

- `defineColumnType` method

  Most of the variants of the `defineColumnType` method are deprecated. The supported variants are for:

  – LOB to LONG conversions
  – Configure the LOB prefetch size

  See the JDBC Javadoc for more information.

- `CONNECTION_PROPERTY_STREAM_CHUNK_SIZE` property

  See the JDBC Javadoc for more information.

- Oracle Update Batching

  Oracle update batching is deprecated from this release in favor of standard update batching.

  See "Standard Update Batching" section on page 21-8 for more information.

- `EndToEndMetrics` related APIs

  `EndToEndMetrics` related APIs are deprecated in this release.

[1] If you are a global user in a PDB, then you cannot perform many tasks without a password-protected role.

See Chapter 33, "JDBC DMS Metrics" for more information.

Desupported Features

The following features are no longer supported by Oracle. See *Oracle Database Upgrade Guide* for a complete list of desupported features in this release.

- Implicit Connection Caching

  See Also: *Oracle Database Upgrade Guide* to see a list of all desupported features in this release of Oracle Database

Other Changes

The following are additional changes in the release:

- Run-Time Connection Load Balancing

  The Run-Time Connection Load Balancing feature enables routing of work requests to an instance that offers the best performance, minimizing the need to relocate work. With the desupport of Implicit Connection Cache, Oracle recommends you to use this feature with Universal Connection Pool. So, the Run-Time Connection Load Balancing chapter is removed from the book from this release.

  See *Oracle Universal Connection Pool for JDBC Developer's Guide* for more information about this feature.

- Fast Connection Failover

  Fast Connection Failover offers a driver-independent way for your Java Database Connectivity (JDBC) application to take advantage of the connection failover facilities offered by Oracle Database 12*c* Release 1 (12.1). With the desupport of Implicit Connection Cache, Oracle recommends you to use this feature with Universal Connection Pool. So, the Fast Connection Failover chapter is removed from the book from this release.

  See *Oracle Universal Connection Pool for JDBC Developer's Guide* for more information about this feature.

Part I
Overview

The chapters in this part introduce the concept of Java Database Connectivity (JDBC) and provide an overview of the Oracle implementation of JDBC. This part provides basic information about installation and configuration of the Oracle client with reference to JDBC drivers. This part also covers the basic steps in creating and running any JDBC application.

Part I contains the following chapters:

- Chapter 1, "Introducing JDBC"
- Chapter 2, "Getting Started"

See Also: *Oracle Database JDBC Java API Reference*

1

Introducing JDBC

Java Database Connectivity (JDBC) is a Java standard that provides the interface for connecting from Java to relational databases. The JDBC standard is defined and implemented through the standard `java.sql` interfaces. This enables individual providers to implement and extend the standard with their own JDBC drivers. JDBC is based on the X/Open SQL Call Level Interface (CLI). JDBC 4.0 complies with the SQL 2003 standard.

This chapter provides an overview of the Oracle implementation of JDBC, covering the following topics:

- Overview of Oracle JDBC Drivers
- Environments and Support
- Feature List

Overview of Oracle JDBC Drivers

In addition to supporting the standard JDBC application programming interfaces (APIs), Oracle drivers have extensions to support Oracle-specific data types and to enhance performance.

Oracle provides the following JDBC drivers:

- Thin driver

  The JDBC Thin driver is a pure Java, Type IV driver that can be used in applications and applets. It is platform-independent and does not require any additional Oracle software on the client-side. The JDBC Thin driver communicates with the server using Oracle Net Services to access Oracle Database.

  The JDBC Thin driver enables a direct connection to the database by providing an implementation of Oracle Net Services on top of Java sockets. The driver supports the TCP/IP protocol and requires a TNS listener on the TCP/IP sockets on the database server.

  Note: Oracle recommends you to use the Thin driver unless you have a feature that is supported only by a specific driver.

  See Also: Chapter 5, "Features Specific to JDBC Thin"

- Oracle Call Interface (OCI) driver

It is used on the client-side with an Oracle client installation. It can be used only with applications.

The JDBC OCI driver is a Type II driver used with Java applications. It requires platform-specific OCI libraries. It supports all installed Oracle Net adapters, including interprocess communication (IPC), named pipes, TCP/IP, and Internetwork Packet Exchange/Sequenced Packet Exchange (IPX/SPX).

The JDBC OCI driver, written in a combination of Java and C, converts JDBC invocations to calls to OCI, using native methods to call C-entry points. These calls communicate with the database using Oracle Net Services.

The JDBC OCI driver uses the OCI libraries, C-entry points, Oracle Net, core libraries, and other necessary files on the client computer where it is installed.

OCI is an API that enables you to create applications that use the native procedures or function calls of a third-generation language to access Oracle Database and control all phases of the SQL statement processing.

See Also: Chapter 6, "Features Specific to JDBC OCI Driver"

- Server-side Thin driver

  It is functionally similar to the client-side Thin driver. However, it is used for code that runs on the database server and needs to access another session either on the same server or on a remote server on any tier.

  The JDBC server-side Thin driver offers the same functionality as the JDBC Thin driver that runs on the client-side. However, the JDBC server-side Thin driver runs inside Oracle Database and accesses a remote database or a different session on the same database for use with Java in the database.

  This driver is useful in the following scenarios:

  - Accessing a remote database server from an Oracle Database instance acting as a middle tier
  - Accessing an Oracle Database session from inside another, such as from a Java stored procedure

  The use of JDBC Thin driver from a client application or from inside a server does not affect the code.

  See Also: Chapter 5, "Features Specific to JDBC Thin"

- Server-side internal driver

  It is used for code that runs on the database server and accesses the same session. That is, the code runs and accesses data from a single Oracle session.

  The JDBC server-side internal driver supports any Java code that runs inside Oracle Database, such as in a Java stored procedure, and accesses the same database. It lets the Oracle Java Virtual Machine (Oracle JVM) to communicate directly with the SQL engine for use with Java in the database.

  The JDBC server-side internal driver, the Oracle JVM, the database, and the SQL engine all run within the same address space, and therefore, the issue of network round-trips is irrelevant. The programs access the SQL engine by using function calls.

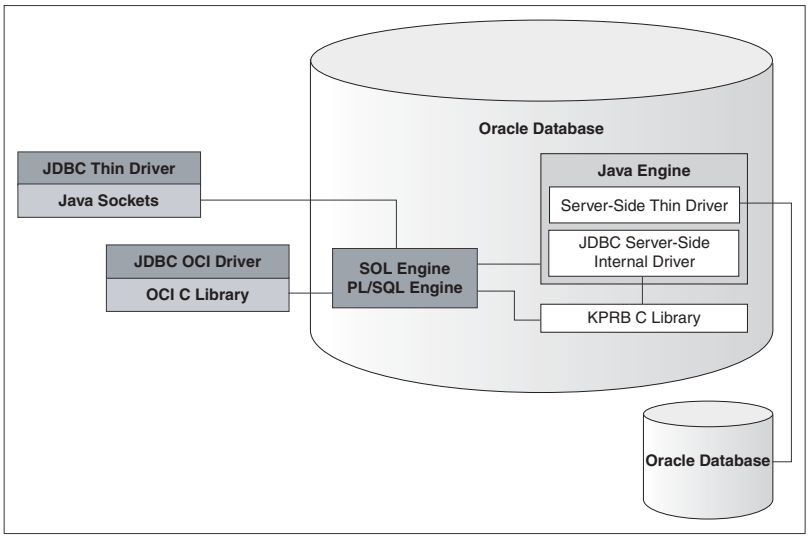
Note: The server-side internal driver does not support the `cancel` and `setQueryTimeout` methods of the `Statement` class.

The JDBC server-side internal driver is fully consistent with the client-side drivers and supports the same features and extensions.

See Also: Chapter 7, "Server-Side Internal Driver"

Figure 1–1 illustrates the architecture of Oracle JDBC drivers and Oracle Database.

Figure 1–1 Architecture of Oracle JDBC Drivers and Oracle Database



This section covers the following topics:

- Choosing the Appropriate Driver
- Feature Differences Between JDBC OCI and Thin Drivers

Choosing the Appropriate Driver

Consider the following when choosing a JDBC driver for your application or applet:

- In general, unless you need OCI-specific features, such as support for non-TCP/IP networks, use the JDBC Thin driver.
- If you want maximum portability and performance, then use the JDBC Thin driver. You can connect to Oracle Database from either an application or an applet using the JDBC Thin driver.
- If you want to use Lightweight Directory Access Protocol (LDAP) over Secure Sockets Layer (SSL), then use the JDBC Thin driver.
- If you are writing a client application for an Oracle client environment and need OCI-driver-specific features, such as support for non-TCP/IP networks, then use the JDBC OCI driver.
- If you are writing an applet, then you must use the JDBC Thin driver.

- For code that runs in the database server and needs to access a remote database or another session within the same database instance, use the JDBC server-side Thin driver.
- If your code runs inside the database server and needs to access data locally within the session, then use the JDBC server-side internal driver to access that server.

Feature Differences Between JDBC OCI and Thin Drivers

Table 1–1 lists the features that are specific either to the JDBC OCI or JDBC Thin driver in Oracle Database 12*c* Release 1 (12.1).

Table 1–1 Feature Differences Between JDBC OCI and JDBC Thin Drivers

JDBC OCI Driver	JDBC Thin Driver
OCI connection pooling	
	Default support for Native XA
Transparent Application Failover (TAF)	
OCI Client Result Cache	
	Application Continuity
	Transaction Guard
	Support for row count per iteration for array DML
	SHA-2 Support in Oracle Advanced Security
`oraaccess.xml` configuration file settings	
	Oracle Advanced Queuing
	Continuous Query Notification
	Support for the `O7L_MR` client ability
	Support for promoting a local transaction to a global transaction

Note:

- The OCI optimized fetch and client-side object cache features are internal to the JDBC OCI driver and are not applicable to the JDBC Thin driver.
- Some JDBC OCI driver features, inherited from the OCI library, are not available in the Thin JDBC driver.

Environments and Support

This section provides a brief discussion of the following topics:

- Supported JDK and JDBC Versions
- JNI and Java Environments
- JDBC and IDEs

Supported JDK and JDBC Versions

In Oracle Database 12*c* Release 1 (12.1), all the JDBC drivers are compatible with JDK 6.0. The JDBC Thin and OCI drivers also support JDK 6. All versions of JDK earlier than 6.0 are no longer supported. Support for JDK 6.0 and JDK 7 is provided through the `ojdbc6.jar` and `ojdbc7.jar` files, respectively.

See Also: "Version Compatibility for Oracle JDBC Drivers" on page 2-1

JNI and Java Environments

The JDBC OCI driver uses the standard Java Native Interface (JNI) to call OCI C libraries. You can use the JDBC OCI driver with Java Virtual Machines (JVMs), in particular, with Microsoft and IBM JVMs.

JDBC and IDEs

The Oracle JDeveloper Suite provides developers with a single, integrated set of products to build, debug, and deploy component-based database applications for the Internet. The Oracle JDeveloper environment contains integrated support for JDBC, including the JDBC Thin driver and the native OCI driver. The database component of Oracle JDeveloper uses the JDBC drivers to manage the connection between the application running on the client and the server.

Feature List

Table 1–2 lists the features and the versions in which they were first supported for each of the three Oracle JDBC drivers: server-side internal driver, JDBC OCI driver, and JDBC Thin driver.

Table 1–2 Feature List

Feature	Server-Side Internal	JDBC OCI	JDBC Thin
JDK 1.0		7.2.2	7.2.2
JDBC 1.0.2		7.2.2	7.2.2
JDK 1.1.1		8.0.6	8.0.6
JDBC 1.22 (No new features; just minor revisions)		8.0.6	8.0.6
defineColumnType[1]		8.0.6	8.0.6
Row Prefetch		8.0.6	8.0.6
Java Native Interface		8.1.6	
JDK 1.2	9.0.1	8.1.6	8.1.6
JDBC 2.0 SQL3 Types (`BLOB`, `CLOB`, `Struct`, `Array`, `REF`)	8.1.5	8.1.5	8.1.5
Native LOB		8.1.6	9.2.0
Associative Arrays[2]	10.2.0	8.1.6	10.1.0
JDBC 2.0 Scrollable Result Sets	8.1.6	8.1.6	8.1.6
JDBC 2.0 Updatable Result Sets	8.1.6	8.1.6	8.1.6
JDBC 2.0 Standard Batching	8.1.6	8.1.6	8.1.6

Table 1–2 (Cont.) Feature List

Feature	Server-Side Internal	JDBC OCI	JDBC Thin
JDBC 2.0 Connection Pooling	NA	8.1.6	8.1.6
JDBC 2.0 XA	8.1.6	8.1.6	8.1.6
Server-side Thin driver	8.1.6	NA	NA
JDBC 2.0 RowSets		9.0.1	9.0.1
Implicit Statement Caching	8.1.7	8.1.7	8.1.7
Explicit Statement Caching	8.1.7	8.1.7	8.1.7
Temporary LOBs	9.0.1	9.0.1	9.0.1
Object Type Inheritance	9.0.1	9.0.1	9.0.1
Multilevel Collections	9.0.1	9.0.1	9.0.1
`oracle.jdbc` Interfaces	9.0.1	9.0.1	9.0.1
Native XA		9.0.1	10.1.0
OCI Connection Pooling	NA	9.0.1	NA
TAF	NA	9.0.1	NA
NLS Support	9.0.1	9.0.1	9.0.1
JDK 1.3	9.2.0	9.2.0	9.2.0
JDK 1.4	10.1.0	9.2.0	9.2.0
JDBC 3.0 Savepoints	9.2.0	9.2.0	9.2.0
New Statement Caching API	9.2.0	9.2.0	9.2.0
ConnectionCacheImpl connection cache	NA	8.1.7	8.1.7
Implicit Connection Cache	NA	10.1.0	10.1.0
Fast Connection Failover		10.1.0.3	10.1.0.3
Connection Wrapping		9.2.0	9.2.0
DMS		9.2.0	9.2.0
Service Names in URLs		9.2.0	10.2.0
JDBC 3.0 Connection Pooling Properties	NA	10.1.0	10.1.0
JDBC 3.0 Updatable BLOB, CLOB, REF	10.1.0	10.1.0	10.1.0
JDBC 3.0 Multiple Open Result Sets	10.1.0	10.1.0	10.1.0
JDBC 3.0 Parameter Metadata	10.1.0	10.1.0	10.1.0
JDBC 3.0 Set/Get Stored Procedures Parameters by Name	10.1.0	10.1.0	10.1.0
JDBC 3.0 Statement Pooling	10.1.0	10.1.0	10.1.0
Set Statement Parameters by Name	10.1.0	10.1.0	10.1.0
End-to-End Tracing		10.1.0	10.1.0
Web RowSet	11.1	10.1.0	10.1.0
Proxy Authentication		10.2.0	10.1.0
JDBC 3.0 Auto Generated Keys		10.2.0	10.2.0
JDBC 3.0 Holdable Cursors	10.2.0	10.2.0	10.2.0

Table 1–2 (Cont.) Feature List

Feature	Server-Side Internal	JDBC OCI	JDBC Thin
JDBC 3.0 Local/Global Transaction Switching	9.2.0	9.2.0	9.2.0
Run-time Connection Load Balancing	NA	10.2.0	10.2.0
Extended `set`*`XXX`* and `get`*`XXX`* for LOBs		10.2.0	10.2.0
XA Connection Cache	NA	10.2.0	10.2.0
DML Returning		10.2.0	10.2.0
JSR 114 RowSets		10.2.0	10.2.0
SSL Encryption		9.2.0	10.2.0
SSL Authentication		9.2.0	11.1
JDK 5.0	11.1	11.1	11.1
JDK 6		11.1	11.1
JDBC 4.0		11.1	11.1
AES Encryption			11.1
SHA1 Hash			11.1
Radius Authentication		10.2.0	11.1
Kerberos Authentication			11.1
ANYDATA and ANYTYPE types		11.1	11.1
Native AQ			11.1
Query Change Notification			11.1
Database startup and shutdown	NA	11.1	11.1
Factory methods for data types	11.1	11.1	11.1
Buffer Cache	11.1	11.1	11.1
Secure Files	11.1	11.1	11.1
Diagnosability	11.1	11.1	11.1
OCI Client Result Cache		11.1.0	
Server Result Cache	11.1	11.1.0	11.1.0
Universal Connection Pool		11.1.0.7.0	11.1.0.7.0
TimeZone Patching		11.2	11.2
Secure Lob Support		11.2	11.2
Lob prefetch Support		11.2	11.2
Network Connection Pool			11.2
Column Security Suppor			11.2
XMLType Queue Support (AQ)			11.2
Notification Grouping (AQ and DCN)			11.2
SimpleFAN		11.2	11.2
Application Continuity			12.1
Transaction Guard			12.1

Table 1–2 (Cont.) Feature List

Feature	Server-Side Internal	JDBC OCI	JDBC Thin
SQL Statement Translation			12.1
Database Resident Connection Pooling		12.1	12.1
Latest JDBC Standard Support		12.1	12.1
SHA-2 Support in Oracle Advanced Security			12.1
Invisible Columns Support		12.1	12.1
Support for PL/SQL Package Types as Parameters		12.1	12.1
Support for Monitoring of Database Operations		12.1	12.1
Support for Increased Length Limit for Various Data Types		12.1	12.1
Implicit Results Support		12.1	12.1
Support for row count per iteration for array DML			12.1
`oraaccess.xml` configuration file settings		12.1	

[1] Starting from Oracle Database 12*c* Release 1 (12.1), most of the variants of this method have been deprecated. The current versions only enable to perform LOB to LONG conversions and configure the LOB prefetch size.

[2] Associative Arrays were previously known as index-by tables.

Note:

- In the table, NA means that the feature is not applicable for the corresponding Oracle JDBC driver.
- The `ConnectionCacheImpl` connection cache feature is deprecated since Oracle Database 10*g*.
- The Implicit Connection Cache feature is desupported from this release.

2

Getting Started

This chapter discusses the compatibility of Oracle Java Database Connectivity (JDBC) driver versions, database versions, and Java Development Kit (JDK) versions. It also describes the basics of testing a client installation and configuration and running a simple application. This chapter contains the following sections:

Version Compatibility for Oracle JDBC Drivers

This section discusses the general JDBC version compatibility issues.

Backward Compatibility

Oracle Database 12*c* Release 1 (12.1) JDBC drivers are certified with supported Oracle Database releases (11.x.0.x). However, they are not certified to work with older, unsupported database releases, such as 10.2.x, 10.1.x, 9.2.x, and 9.0.1.x.

Forward Compatibility

Existing and supported JDBC drivers are certified to work with Oracle Database 12*c* Release 1 (12.1).

Note:

- In Oracle Database 12*c* Release 1 (12.1), Oracle JDBC drivers no longer support JDK 1.4.x or earlier versions.
- You can find a complete, up-to-date list of supported databases at `http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-faq-090281.html`

Verifying a JDBC Client Installation

To verify a JDBC client installation, you must do all of the following:

This section describes the steps for verifying an Oracle client installation of the JDBC drivers, assuming that you have already installed the driver of your choice. Installation of an Oracle JDBC driver is platform-specific. You must follow the installation instructions for the driver you want to install in your platform-specific documentation.

If you use the JDBC Thin driver, then there is no additional installation on the client computer. If you use the JDBC Oracle Call Interface (OCI) driver, then you must also install the Oracle client software. This includes Oracle Net and the OCI libraries.

> **Note:** The JDBC Thin driver requires a TCP/IP listener to be running on the computer where the database is installed.

Check the Installed Directories and Files

Installing the Oracle Java products creates, among other things, the following directories:

- *ORACLE_HOME*/jdbc
- *ORACLE_HOME* /jlib

Check whether or not the following directories and files have been created and populated in the *ORACLE_HOME*/jdbc directory:

- demo

  This directory contains a compressed file, demo.zip or demo.tar. When you uncompress this compressed file, the samples directory and the Samples-Readme.txt file are created. The samples directory contains sample programs, including examples of how to use JDBC escape syntax and Oracle SQL syntax, PL/SQL blocks, streams, user-defined types, additional Oracle type extensions, and Oracle performance extensions.

- doc

  This directory contains the javadoc.zip file, which is the Oracle JDBC application programming interface (API) documentation.

- lib

  The lib directory contains the following required Java classes:

  – orai18n.jar and orai18n-mapping.jar

    Contain classes for globalization and multibyte character sets support

  – ojdbc6.jar, ojdbc6_g.jar, ojdbc7.jar, and ojdbc7_g.jar

    Contain the JDBC driver classes for use with JDK 6 and JDK 7

Note:

- Since Oracle Database 11*g* Release 1, support for a version of JDK earlier than version 5.0 has been removed. Also, the `ojdbc14.jar`, `ojdbc5.jar` and `classes12.jar` files are no longer shipped. Instead, you can use the `ojdbc6.jar` and `ojdbc7.jar` files, which are shipped with Oracle Database 12*c*.
- If you are using JSE 6 and later, then there is *no* need to explicitly load the JDBC driver. This means that the Java run-time loads the driver when needed and you need *not* include `Class.forName("oracle.jdbc.OracleDriver")` or `new oracle.jdbc.OracleDriver()` in your code. But if you are using J2SE 5.0, then you need to load the JDBC driver explicitly.

- `Readme.txt`

  This file contains late-breaking and release-specific information about the drivers, which may not have been included in other documentation on the product.

Check whether or not the following directories have been created and populated in the *ORACLE_HOME*`/jlib` directory:

- `jta.jar` and `jndi.jar`

  These files contain classes for the Java Transaction API (JTA) and the Java Naming and Directory Interface (JNDI). These are required only if you are using JTA features for distributed transaction management or JNDI features for naming services.

  Note: For more information about these files, visit the following sites

  `http://www.oracle.com/technetwork/java/javaee/jta/index.html`

  `http://www.oracle.com/technetwork/java/jndi/index.html`

- `ons.jar`

  This JAR file contains classes for Oracle RAC Fast Application Notification. It is also required for Universal Connection Pool (UCP) features like Fast Connection Failover, Run-time Load Balancing, Web Session Affinity, and Transaction Affinity.

  See Also: Appendix B, "Oracle RAC Fast Application Notification" and *Oracle Universal Connection Pool for JDBC Developer's Guide* for more information about Oracle RAC Fast Application Notification and UCP respectively

Check the Environment Variables

This section describes the environment variables that must be set for the JDBC OCI driver and the JDBC Thin driver, focusing on Solaris, Linux, and Microsoft Windows platforms.

You must set the `CLASSPATH` environment variable for JDBC OCI or Thin driver. Include the following in the `CLASSPATH` environment variable:

```
ORACLE_HOME/jdbc/lib/ojdbc6.jar
ORACLE_HOME/jlib/orai18n.jar
```

Note: If you use the JTA features and the JNDI features, then you must specify `jta.jar` and `jndi.jar` in your `CLASSPATH` environment variable.

JDBC OCI Driver

To use the JDBC OCI driver, you must also set the following value for the library path environment variable:

- On Solaris or Linux, set the `LD_LIBRARY_PATH` environment variable as follows:

  ORACLE_HOME`/lib`

  This directory contains the `libocijdbc11.so` shared object library.

- On Microsoft Windows, set the `PATH` environment variable as follows:

  ORACLE_HOME`\bin`

  This directory contains the `ocijdbc11.dll` dynamic link library.

All of the JDBC OCI demonstration programs can be run in the Instant Client mode by including the JDBC OCI Instant Client data shared library on the library path environment variable.

See Also: Chapter 6, "Features Specific to JDBC OCI Driver"

JDBC Thin Driver

To use the JDBC Thin driver, you do not have to set any other environment variables. However, to use the JDBC server-side Thin driver, you need to set permission.

Setting Permission for the Server-Side Thin Driver

The JDBC server-side Thin driver opens a socket for its connection to the database. Because Oracle Database enforces the Java security model, a check is performed for a `SocketPermission` object.

To use the JDBC server-side Thin driver, the connecting user must be granted the appropriate permission. The following is an example of how the permission can be granted for the user `HR`:

```
CREATE ROLE jdbcthin;
CALL dbms_java.grant_permission('JDBCTHIN', 'java.net.SocketPermission', '*',
'connect');
GRANT jdbcthin TO HR;
```

Note that `JDBCTHIN` in the `grant_permission` call must be in uppercase. The asterisk (`*`) is a pattern. You can restrict the user by granting permission to connect to only specific computers or ports.

See Also: *Oracle Database Java Developer's Guide*

Ensure that the Java Code Can Be Compiled and Run

To further ensure that Java is set up properly on your client system, go to the `samples` directory under the *ORACLE_HOME*`/jdbc/demo` directory. Now, type the following commands on the command line, one after the other, to see if the Java compiler and the Java interpreter run without error:

```
javac
```

```
java
```

Each of the preceding commands should display a list of options and parameters and then exit. Ideally, verify that you can compile and run a simple test program, such as `jdbc/demo/samples/generic/SelectExample`.

Determine the Version of the JDBC Driver

To determine the version of the JDBC driver, call the `getDriverVersion` method of the `OracleDatabaseMetaData` class as shown in the following sample code:

```
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;

class JDBCVersion
{
  public static void main (String args[]) throws SQLException
  {
    OracleDataSource ods = new OracleDataSource();
    ods.setURL("jdbc:oracle:thin:HR/hr@<host>:<port>:<service>");
    Connection conn = ods.getConnection();

    // Create Oracle DatabaseMetaData object
    DatabaseMetaData meta = conn.getMetaData();

    // gets driver info:
    System.out.println("JDBC driver version is " + meta.getDriverVersion());
  }
}
```

You can also determine the version of the JDBC driver by executing the following commands:

- `java -jar ojdbc6.jar`
- `java -jar ojdbc7.jar`

Test JDBC and the Database Connection

The `samples` directory contains sample programs for a particular Oracle JDBC driver. One of the programs, `JdbcCheckup.java`, is designed to test JDBC and the database connection. The program queries for the user name, password, and the name of the database to which you want to connect. The program connects to the database, queries for the string "`Hello World`", and prints it to the screen.

Go to the `samples` directory, and compile and run the `JdbcCheckup.java` program. If the results of the query print without error, then your Java and JDBC installations are correct.

Although `JdbcCheckup.java` is a simple program, it demonstrates several important functions by performing the following:

- Imports the necessary Java classes, including JDBC classes
- Creates a `DataSource` instance
- Connects to the database
- Runs a simple query

- Prints the query results to your screen

The `JdbcCheckup.java` program, which uses the JDBC OCI driver, is as follows:

```
/*
 * This sample can be used to check the JDBC installation.
 * Just run it and provide the connect information. It will select
 * "Hello World" from the database.
 */

// You need to import the java.sql and JDBC packages to use JDBC
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;

// We import java.io to be able to read from the command line
import java.io.*;

class JdbcCheckup
{
  public static void main(String args[]) throws SQLException, IOException
  {

    // Prompt the user for connect information
    System.out.println("Please enter information to test connection to
                        the database");
    String user;
    String password;
    String database;

    user = readEntry("user: ");
    int slash_index = user.indexOf('/');
    if (slash_index != -1)
    {
      password = user.substring(slash_index + 1);
      user = user.substring(0, slash_index);
    }
    else
      password = readEntry("password: ");
    database = readEntry("database(a TNSNAME entry): ");

    System.out.print("Connecting to the database...");
    System.out.flush();
    System.out.println("Connecting...");
    // Open an OracleDataSource and get a connection
    OracleDataSource ods = new OracleDataSource();
    ods.setURL("jdbc:oracle:oci:@" + database);
    ods.setUser(user);
    ods.setPassword(password);
    Connection conn = ods.getConnection();
    System.out.println("connected.");

    // Create a statement
    Statement stmt = conn.createStatement();

    // Do the SQL "Hello World" thing
    ResultSet rset = stmt.executeQuery("select 'Hello World' from dual");

    while (rset.next())
      System.out.println(rset.getString(1));
    // close the result set, the statement and the connection
```

```
    rset.close();
    stmt.close();
    conn.close();
    System.out.println("Your JDBC installation is correct.");
  }

  // Utility function to read a line from standard input
  static String readEntry(String prompt)
  {
    try
    {
      StringBuffer buffer = new StringBuffer();
      System.out.print(prompt);
      System.out.flush();
      int c = System.in.read();
      while (c != '\n' && c != -1)
      {
        buffer.append((char)c);
        c = System.in.read();
      }
      return buffer.toString().trim();
    }
    catch(IOException e)
    {
      return "";
    }
  }
}
```

Basic Steps in JDBC

After verifying the JDBC client installation, you can start creating your JDBC applications. When using Oracle JDBC drivers, you must include certain driver-specific information in your programs. This section describes, in the form of a tutorial, where and how to add the information. The tutorial guides you through the steps to create code that connects to and queries a database from the client.

You must write code to perform the following tasks:

1. Importing Packages
2. Opening a Connection to a Database
3. Creating a Statement Object
4. Running a Query and Retrieving a Result Set Object
5. Processing the Result Set Object
6. Closing the Result Set and Statement Objects
7. Making Changes to the Database
8. Committing Changes
9. Closing the Connection

Note: You must supply Oracle driver-specific information for the first three tasks that enable your program to use the JDBC application programming interface (API) to access a database. For the other tasks, you can use standard JDBC Java code, as you would for any Java application.

Importing Packages

Regardless of which Oracle JDBC driver you use, include the `import` statements shown in Table 2–1 at the beginning of your program.

Table 2–1 Import Statements for JDBC Driver

Import statement	Provides
`import java.sql.*;`	Standard JDBC packages.
`import java.math.*;`	The `BigDecimal` and `BigInteger` classes. You can omit this package if you are not going to use these classes in your application.
`import oracle.jdbc.*;`	Oracle extensions to JDBC. This is optional.
`import oracle.jdbc.pool.*;`	`OracleDataSource`.
`import oracle.sql.*;`	Oracle type extensions. This is optional.

The Oracle packages listed as optional provide access to the extended functionality provided by Oracle JDBC drivers, but are not required for the example presented in this section.

Note: It is better to import only the classes your application needs, rather than using the wildcard asterisk (`*`). This guide uses the asterisk (*) for simplicity, but this is not the recommended way of importing classes and interfaces.

Opening a Connection to a Database

First, you must create an `OracleDataSource` instance. Then, open a connection to the database using the `OracleDataSource.getConnection` method. The properties of the retrieved connection are derived from the `OracleDataSource` instance. If you set the URL connection property, then all other properties, including `TNSEntryName`, `DatabaseName`, `ServiceName`, `ServerName`, `PortNumber`, `Network Protocol`, and driver type are ignored.

Specifying a Database URL, User Name, and Password

The following code sets the URL, user name, and password for a data source:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setUser(user);
ods.setPassword(password);
```

The following example connects user `HR` with password `hr` to a database with service `orcl` through port 5221 of the host `myhost`, using the JDBC Thin driver:

```
OracleDataSource ods = new OracleDataSource();
String url = "jdbc:oracle:thin:@//myhost:5221/orcl";
```

```
ods.setURL(url);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

Note: The user name and password specified in the arguments override any user name and password specified in the URL.

Specifying a Database URL that Includes User Name and Password

The following example connects user `HR` with password `hr` to a database host whose Transparent Network Substrate (TNS) entry is `myTNSEntry`, using the JDBC Oracle Call Interface (OCI) driver. In this case, the URL includes the user name and password and is the only input parameter.

```
String url = "jdbc:oracle:oci:HR/hr@myTNSEntry");
ods.setURL(url);
Connection conn = ods.getConnection();
```

If you want to connect using the Thin driver, then you must specify the port number. For example, if you want to connect to the database on the host `myhost` that has a TCP/IP listener on port 5221 and the service identifier is `orcl`, then provide the following code:

```
String URL = "jdbc:oracle:thin:HR/hr@//myhost:5221/orcl");
ods.setURL(URL);
Connection conn = ods.getConnection();
```

See Also: Chapter 8, "Data Sources and URLs"

Creating a Statement Object

Once you connect to the database and, in the process, create a `Connection` object, the next step is to create a `Statement` object. The `createStatement` method of the JDBC `Connection` object returns an object of the JDBC `Statement` type. To continue the example from the previous section, where the `Connection` object `conn` was created, here is an example of how to create the `Statement` object:

```
Statement stmt = conn.createStatement();
```

Running a Query and Retrieving a Result Set Object

To query the database, use the `executeQuery` method of the `Statement` object. This method takes a SQL statement as input and returns a JDBC `ResultSet` object.

Note:

- The method used to execute a `Statement` object depends on the type of SQL statement being executed. If the `Statement` object represents a SQL query returning a `ResultSet` object, the `executeQuery` method should be used. If the SQL is known to be a DDL statement or a DML statement returning an update count, the `executeUpdate` method should be used. If the type of the SQL statement is not known, the `execute` method should be used.
- In case of a standard JDBC driver, if the SQL string being executed does not return a `ResultSet` object, then the `executeQuery` method throws a `SQLException` exception. In case of an Oracle JDBC driver, the `executeQuery` method does not throw a `SQLException` exception even if the SQL string being executed does not return a `ResultSet` object.

To continue the example, once you create the `Statement` object `stmt`, the next step is to run a query that returns a `ResultSet` object with the contents of the `first_name` column of a table of employees named `EMPLOYEES`:

```
ResultSet rset = stmt.executeQuery ("SELECT first_name FROM employees");
```

Processing the Result Set Object

Once you run your query, use the `next()` method of the `ResultSet` object to iterate through the results. This method steps through the result set row by row, detecting the end of the result set when it is reached.

To pull data out of the result set as you iterate through it, use the appropriate `get`*`XXX`* methods of the `ResultSet` object, where *`XXX`* corresponds to a Java data type.

For example, the following code will iterate through the `ResultSet` object, `rset`, from the previous section and will retrieve and print each employee name:

```
while (rset.next())
   System.out.println (rset.getString(1));
```

The `next()` method returns `false` when it reaches the end of the result set. The employee names are materialized as Java `String` values.

Closing the Result Set and Statement Objects

You must explicitly close the `ResultSet` and `Statement` objects after you finish using them. This applies to all `ResultSet` and `Statement` objects you create when using Oracle JDBC drivers. The drivers do not have finalizer methods. The cleanup routines are performed by the `close` method of the `ResultSet` and `Statement` classes. If you do not explicitly close the `ResultSet` and `Statement` objects, serious memory leaks could occur. You could also run out of cursors in the database. Closing both the result set and the statement releases the corresponding cursor in the database. If you close only the result set, then the cursor is not released.

For example, if your `ResultSet` object is `rset` and your `Statement` object is `stmt`, then close the result set and statement with the following lines of code:

```
rset.close();
stmt.close();
```

When you close a `Statement` object that a given `Connection` object creates, the connection itself remains open.

> **Note:** Typically, you should put `close` statements in a `finally` clause.

Making Changes to the Database

DML Operations

To perform DML (Data Manipulation Language) operations, such as INSERT or UPDATE operations, you can create either a `Statement` object or a `PreparedStatement` object. `PreparedStatement` objects enable you to run a statement with varying sets of input parameters. The `prepareStatement` method of the JDBC `Connection` object lets you define a statement that takes variable bind parameters and returns a JDBC `PreparedStatement` object with your statement definition.

Use the `set`*`XXX`* methods on the `PreparedStatement` object to bind data to the prepared statement to be sent to the database.

> **See Also:** "The setObject and setOracleObject Methods" on page 11-11 and "Other setXXX Methods" on page 11-11

The following example shows how to use a prepared statement to run `INSERT` operations that add two rows to the `EMPLOYEES` table.

```
    // Prepare to insert new names in the EMPLOYEES table
PreparedStatement pstmt = null;
try{
    pstmt = conn.prepareStatement ("insert into EMPLOYEES (EMPLOYEE_ID, FIRST_
NAME) values (?, ?)");

    // Add LESLIE as employee number 1500
    pstmt.setInt (1, 1500);          // The first ? is for EMPLOYEE_ID
    pstmt.setString (2, "LESLIE");   // The second ? is for FIRST_NAME
    // Do the insertion
    pstmt.execute();

    // Add MARSHA as employee number 507
    pstmt.setInt (1, 507);           // The first ? is for EMPLOYEE_ID
    pstmt.setString (2, "MARSHA");   // The second ? is for FIRST_NAME
    // Do the insertion
    pstmt.execute();
}

finally{
if(pstmt!=null)

    // Close the statement
    pstmt.close();
}
```

DDL Operations

To perform data definition language (DDL) operations, you must create a `Statement` object. The following example shows how to create a table in the database:

```
//create table EMPLOYEES with columns EMPLOYEE_ID and FIRST_NAME
String query;
Statement stmt=null;

try{
    query="create table EMPLOYEES " +
          "(EMPLOYEE_ID int, " +
          "FIRST_NAME varchar(50))";
    stmt = conn.createStatement();
    stmt.executeUpdate(query);
    }
finally{
     //close the Statement object
     stmt.close();
    }
```

> **Note:** You can also use a `PreparedStatement` object to perform DDL operations. However, you should not use a `PreparedStatement` object because the useful part of such an object is that it can have parameters and a DDL operation does not have any parameters.
>
> Also, due to a Database limitation, if you use a `PreparedStatement` object for a DDL operation, then it only works for the first time it is executed. So, you should use only `Statement` objects for DDL operations.

The following example shows how to prepare your DDL statements before any reexecution:

```
//
Statement stmt = null;
PreparedStatement pstmt = null;
try{
    pstmt = conn.prepareStatement ("insert into EMPLOYEES (EMPLOYEE_ID, FIRST_
NAME) values (?, ?)");
    stmt = conn.createStatement("truncate table EMPLOYEES");

    // Add LESLIE as employee number 1500
    pstmt.setInt (1, 1500);          // The first ? is for EMPLOYEE_ID
    pstmt.setString (2, "LESLIE");   // The second ? is for FIRST_NAME
    pstmt.execute();
    stmt.executeUpdate();

    // Add MARSHA as employee number 507
    pstmt.setInt (1, 507);           // The first ? is for EMPLOYEE_ID
    pstmt.setString (2, "MARSHA");   // The second ? is for FIRST_NAME
    pstmt.execute();
    stmt.executeUpdate();
    }
finally{
if(pstmt!=null)

    // Close the statement
     pstmt.close();
}
```

Committing Changes

By default, data manipulation language (DML) operations are committed automatically as soon as they are run. This is known as the auto-commit mode. If auto-commit mode is on and you perform a `COMMIT` or `ROLLBACK` operation using the `commit` or `rollback` method on a connection object, then you get the following error messages:

Table 2–2 Error Messages for Operations Performed When Auto-Commit Mode is ON

Operation	Error Messages
`COMMIT`	`Could not commit with auto-commit set on`
`ROLLBACK`	`Could not rollback with auto-commit set on`

If a SQLException is raised during a `COMMIT` or `ROLLBACK` operation with the error messages as mentioned in Table 2–2, then check the auto-commit status of the connection because you get an exception when these operations are performed on a connection that has auto-commit value set to `true`.

This exception is raised for any one of the following cases:

- When auto-commit status is set to `true` and `commit` or `rollback` method is called
- When the default status of auto-commit is not changed and `commit` or `rollback` method is called
- When the value of the `COMMIT_ON_ACCEPT_CHANGES` property is `true` and `commit` or `rollback` method is called after calling the `acceptChanges` method on a rowset

However, you can disable auto-commit mode with the following method call on the `Connection` object:

```
conn.setAutoCommit(false);
```

See Also: "Disabling Auto-Commit Mode" on page C-2.

If you disable the auto-commit mode, then you must manually commit or roll back changes with the appropriate method call on the `Connection` object:

```
conn.commit();
```

or:

```
conn.rollback();
```

A `COMMIT` or `ROLLBACK` operation affects all DML statements run since the last `COMMIT` or `ROLLBACK`.

Note:

- If the auto-commit mode is disabled and you close the connection without explicitly committing or rolling back your last changes, then an implicit `COMMIT` operation is run.
- Any data definition language (DDL) operation always causes an implicit `COMMIT`. If the auto-commit mode is disabled, then this implicit `COMMIT` will commit any pending DML operations that had not yet been explicitly committed or rolled back.

Changing Commit Behavior

When a transaction updates the database, it generates a redo entry corresponding to this update. Oracle Database buffers this redo in memory until the completion of the transaction. When you commit the transaction, the Log Writer (LGWR) process writes the redo entry for the commit to disk, along with the accumulated redo entries of all changes in the transaction. By default, Oracle Database writes the redo to disk before the call returns to the client. This behavior introduces latency in the commit because the application must wait for the redo entry to be persisted on disk.

If your application requires very high transaction throughput and you are willing to trade commit durability for lower commit latency, then you can change the behavior of the default `COMMIT` operation, depending on the needs of your application. You can change the behavior of the `COMMIT` operation with the following options:

- `WAIT`
- `NOWAIT`
- `WRITEBATCH`
- `WRITEIMMED`

These options let you control two different aspects of the commit phase:

- Whether the `COMMIT` call should wait for the server to process it or not. This is achieved by using the `WAIT` or `NOWAIT` option.
- Whether the Log Writer should batch the call or not. This is achieved by using the `WRITEIMMED` or `WRITEBATCH` option.

You can also combine different options together. For example, if you want the `COMMIT` call to return without waiting for the server to process it and also the log writer to process the commits in batch, then you can use the `NOWAIT` and `WRITEBATCH` options together. For example:

```
((OracleConnection)conn).commit(
    EnumSet.of(
      OracleConnection.CommitOption.WRITEBATCH,
      OracleConnection.CommitOption.NOWAIT));
```

Note: you cannot use the `WAIT` and `NOWAIT` options together because they have opposite meanings. If you do so, then the JDBC driver will throw an exception. The same applies to the `WRITEIMMED` and `WRITEBATCH` options.

Working with Invisible Columns

Starting from this release, Oracle Database supports invisible columns. Using this feature, you can add a column to the table in hidden mode and make it visible later. JDBC provides APIs to retrieve information about invisible columns. To get information about whether a column is invisible or not, you can use the `isColumnInvisible` method available in the `oracle.jdbc.OracleResultSetMetaData` interface in the following way:

Example

```
...
Connection conn = DriverManager.getConnection(jdbcURL, user, password);
Statement stmt = conn.createStatement ();
stmt.executeQuery ("create table hiddenColsTable (a varchar(20), b int
```

```
invisible)");
stmt.executeUpdate("insert into hiddenColsTable (a,b ) values('somedata',1)");
stmt.executeUpdate("insert into hiddenColsTable (a,b) values('newdata',2)");

System.out.println ("Invisible columns information");
try
{
      ResultSet rset = stmt.executeQuery("SELECT a, b FROM hiddenColsTable");
      OracleResultSetMetaData rsmd = (OracleResultSetMetaData)rset.getMetaData();
      while (rset.next())
      {
        System.out.println("column1 value:" + rset.getString(1));
        System.out.println("Visibility:" + rsmd.isColumnInvisible(1));
        System.out.println("column2 value:" + rset.getInt(2));
        System.out.println("Visibility:" + rsmd.isColumnInvisible(2));
      }
}
catch (Exception ex)
{
       System.out.println("Exception :" + ex);
       ex.printStackTrace();
}
```

Alternatively, you can also use the `getColumns` method available in the `oracle.jdbc.OracleDatabaseMetaData` class to retrieve information about invisible columns.

Example

```
...
Connection conn = DriverManager.getConnection(jdbcURL, user, password);
Statement stmt = conn.createStatement ();
stmt.executeQuery ("create table hiddenColsTable (a varchar(20), b int
invisible)");
stmt.executeUpdate("insert into hiddenColsTable (a,b ) values('somedata',1)");
stmt.executeUpdate("insert into hiddenColsTable (a,b) values('newdata',2)");

 System.out.println ("getColumns for table with invisible columns");
 try
 {
      DatabaseMetaData dbmd = conn.getMetaData();
      ResultSet rs = dbmd.getColumns(null, "HR", "hiddenColsTable", null);
      OracleResultSetMetaData rsmd = (OracleResultSetMetaData)rs.getMetaData();
      int colCount = rsmd.getColumnCount();
      System.out.println("colCount: " + colCount);
      String[] columnNames = new String [colCount];

      for (int i = 0; i < colCount; ++i)
      {
        columnNames[i] = rsmd.getColumnName (i + 1);
      }

      while (rs.next())
      {
        for (int i = 0; i < colCount; ++i)
           System.out.println(columnNames[i] +":" +rs.getString (columnNames[i]));
      }
 }
 catch (Exception ex)
 {
```

```
      System.out.println("Exception: " + ex);
      ex.printStackTrace();
 }
```

Note: The server-side internal driver, `kprb` does not support fetching information about invisible columns.

Closing the Connection

You must close the connection to the database after you have performed all the required operations and no longer require the connection. You can close the connection by using the `close` method of the `Connection` object, as follows:

```
conn.close();
```

Note: Typically, you should put `close` statements in a `finally` clause.

Sample: Connecting, Querying, and Processing the Results

The steps in the preceding sections are illustrated in the following example, which uses the Oracle JDBC Thin driver to create a data source, connects to the database, creates a `Statement` object, runs a query, and processes the result set.

Note that the code for creating the `Statement` object, running the query, returning and processing the `ResultSet` object, and closing the statement and connection uses the standard JDBC API.

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.SQLException;
import oracle.jdbc.pool.OracleDataSource;

class JdbcTest
{
   public static void main (String args []) throws SQLException
   {

OracleDataSource ods = null;
Connection conn = null;
Statement stmt = null;
ResultSet rset = null;

      // Create DataSource and connect to the local database
      ods = new OracleDataSource();
      ods.setURL("jdbc:oracle:thin:@//localhost:5221/orcl");
      ods.setUser("HR");
      ods.setPassword("hr");
      conn = ods.getConnection();

try
{
      // Query the employee names
      stmt = conn.createStatement ();
      rset = stmt.executeQuery ("SELECT first_name FROM employees");
```

```
      // Print the name out
      while (rset.next ())
         System.out.println (rset.getString (1));
    }

      //Close the result set, statement, and the connection

finally{
      if(rset!=null) rset.close();
      if(stmt!=null) stmt.close();
      if(conn!=null) conn.close();
}
   }
}
```

If you want to adapt the code for the OCI driver, then replace the call to the `OracleDataSource.setURL` method with the following:

```
ods.setURL("jdbc:oracle:oci:@MyHostString");
```

where, `MyHostString` is an entry in the `TNSNAMES.ORA` file.

Stored Procedure Calls in JDBC Programs

This section describes how Oracle JDBC drivers support the following kinds of stored procedures:

- PL/SQL Stored Procedures
- Java Stored Procedures
- Support for Implicit Results

PL/SQL Stored Procedures

JDBC supports the invocation of PL/SQL procedures/functions and anonymous blocks, using either JDBC escape syntax or PL/SQL block syntax. The following PL/SQL calls would work with any Oracle JDBC driver:

```
// JDBC escape syntax
CallableStatement cs1 = conn.prepareCall
                        ( "{call proc (?,?)}" ) ; // stored proc
CallableStatement cs2 = conn.prepareCall
                        ( "{? = call func (?,?)}" ) ; // stored func
// PL/SQL block syntax
CallableStatement cs3 = conn.prepareCall
                        ( "begin proc (?,?); end;" ) ; // stored proc
CallableStatement cs4 = conn.prepareCall
                        ( "begin ? := func(?,?); end;" ) ; // stored func
```

As an example of using the Oracle syntax, here is a PL/SQL code snippet that creates a stored function. The PL/SQL function gets a character sequence and concatenates a suffix to it:

```
create or replace function foo (val1 char)
return char as
begin
   return val1 || 'suffix';
end;
```

The function invocation in your JDBC program should look like the following:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci:@<hoststring>");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();

CallableStatement cs = conn.prepareCall ("begin ? := foo(?); end;");
cs.registerOutParameter(1,Types.CHAR);
cs.setString(2, "aa");
cs.execute();
String result = cs.getString(1);
```

Java Stored Procedures

You can use JDBC to call Java stored procedures through the SQL interface. The syntax for calling Java stored procedures is the same as the syntax for calling PL/SQL stored procedures, presuming they have been properly published. That is, you have written call specifications to publish them to the Oracle data dictionary. Applications can call Java stored procedures using the Native Java Interface for direct invocation of `static` Java methods.

Support for Implicit Results

Starting from this release, Oracle Database supports results of SQL statements executed in a stored procedure to be returned implicitly to the client applications without the need to explicitly use a `REF CURSOR`. You can use the following methods to retrieve and process the implicit results returned by PL/SQL procedures or blocks:

Method	Description
`getMoreResults`	Checks if there are more results available in the result set
`getMoreResults(int)`	Checks if there are more results available in the result set, like the overloaded method. This method accepts an `int` parameter that can have one of the following values: ■ `KEEP_CURRENT_RESULT` ■ `CLOSE_ALL_RESULTS` ■ `CLOSE_CURRENT_RESULT`
`getResultSet`	Iteratively retrieves each implicit result from an executed PL/SQL statement

Note:

- The server-side internal driver, `kprb` does not support fetching information about implicit results.
- Only SELECT queries can be returned implicitly.
- Applications retrieve each result set sequentially, but can fetch rows from any result set independent of the sequence.

Suppose you have a procedure called `foo` as the following:

```
create procedure foo as
```

```
  c1 sys_refcursor;
  c2 sys_refcursor;
begin
  open c1 for select * from hr.employees;
  dbms_sql.return_result(c1); --return to client
  -- open 1 more cursor
  open c2 for select * from hr.departments;
  dbms_sql.return_result (c2); --return to client
end;
```

The following code snippet demonstrates how to retrieve the implicit results returned by PL/SQL procedures using the `getMoreResults` methods:

Example 1

```
String sql = "begin foo; end;";
...
Connection conn = DriverManager.getConnection(jdbcURL, user, password);
 try {
        Statement stmt = conn.createStatement ();
        stmt.executeQuery (sql);

        while (stmt.getMoreResults())
        {
              ResultSet rs = stmt.getResultSet();
              System.out.println("ResultSet");
              while (rs.next())
             {
                   /* get results */
             }
         }
      }
```

Suppose you have another procedure called `foo` as the following:

```
create or replace procedure foo as
c1 sys_refcursor;
c2 sys_refcursor;
c3 sys_refcursor;
begin   open c1 for 'select * from hr.employees';
dbms_sql.return_result (c1);
-- cursor 2
open c2 for 'select * from hr.departments';
dbms_sql.return_result (c2);
-- cursor 3
open c3 for 'select first_name from hr.employees';
dbms_sql.return_result (c3);
end;
```

The following code snippet demonstrates how to retrieve the implicit results returned by PL/SQL procedures using the `getMoreResults(int)` methods:

Example 2

```
String sql = "begin foo; end;";
...
Connection conn = DriverManager.getConnection(jdbcURL, user, password);

try {
        Statement stmt = conn.createStatement ();
        stmt.executeQuery (sql);
```

```
        ResultSet rs = null;

        boolean retval = stmt.getMoreResults(Statement.KEEP_CURRENT_RESULT))
        if (retval)
        {
            rs = stmt.getResultSet();
            System.out.println("ResultSet");
            while (rs.next())
            {
                /* get results */
            }
        }

        /* closes open results */
        retval = stmt.getMoreResults(Statement.CLOSE_ALL_RESULTS);

        if (retval)
        {
            System.out.println("More ResultSet available");
            rs = stmt.getResultSet();
            System.out.println("ResultSet");
            while (rs.next())
            {
                /* get results */
            }
        }

        /* close current result set */
        retval = stmt.getMoreResults(Statement.CLOSE_CURRENT_RESULT);

        if(retval)
        {
            System.out.println("More ResultSet available");
            rs = stmt.getResultSet();
            while (rs.next())
            {
                /* get Results */

            }
        }
    }
```

Processing SQL Exceptions

To handle error conditions, Oracle JDBC drivers throw SQL exceptions, producing instances of the `java.sql.SQLException` class or its subclass. Errors can originate either in the JDBC driver or in the database itself. Resulting messages describe the error and identify the method that threw the error. Additional run-time information can also be appended.

JDBC 3.0 defines only a single exception, `SQLException`. However, there are large categories of errors and it is useful to distinguish them. Therefore, in JDBC 4.0, a set of subclasses of the `SQLException` exception is introduced to identify the different categories of errors. To know more about this feature, see Support for JDBC 4.0 Standard on page 3-6.

Basic exception handling can include retrieving the error message, retrieving the error code, retrieving the SQL state, and printing the stack trace. The `SQLException` class includes functionality to retrieve all of this information, when available.

See Also:

- Appendix D, "JDBC Error Messages"
- *Oracle Database Error Messages*

Retrieving Error Information

You can retrieve basic error information with the following methods of the `SQLException` class:

- `getMessage`
- `getErrorCode`
- `getSQLState`

The following example prints output from a `getMessage` method call:

```
catch(SQLException e)
{
   System.out.println("exception: " + e.getMessage());
}
```

This would print the output, such as the following, for an error originating in the JDBC driver:

```
exception: Invalid column type
```

Note: Error message text is available in alternative languages and character sets supported by Oracle.

Printing the Stack Trace

The `SQLException` class provides the `printStackTrace()` method for printing a stack trace. This method prints the stack trace of the `Throwable` object to the standard error stream. You can also specify a `java.io.PrintStream` object or `java.io.PrintWriter` object for output.

The following code fragment illustrates how you can catch SQL exceptions and print the stack trace.

```
try { <some code> }
catch(SQLException e) { e.printStackTrace (); }
```

To illustrate how the JDBC drivers handle errors, assume the following code uses an incorrect column index:

```
// Iterate through the result and print the employee names
// of the code

try {
  while (rset.next ())
      System.out.println (rset.getString (5));  // incorrect column index
}
catch(SQLException e) { e.printStackTrace (); }
```

Assuming the column index is incorrect, running the program would produce the following error text:

```
java.sql.SQLException: Invalid column index
at
oracle.jdbc.OracleDriver.OracleResultSetImpl.getDate(OracleResultSetImpl.java:1556
)
at Employee.main(Employee.java:41)
```

Part II

Oracle JDBC

This part includes chapters that discuss the different Java Database Connectivity (JDBC) versions that Oracle Database 12*c* supports. It also includes chapters that cover features specific to JDBC Thin driver, JDBC Oracle Call Interface (OCI) driver, and the server-side internal driver.

Part II contains the following chapters:

3

JDBC Standards Support

Oracle Java Database Connectivity (JDBC) drivers support different versions of the JDBC standard features. In Oracle Database 12*c* Release 1 (12.1), Oracle JDBC drivers have been enhanced to provide support for the JDBC 4.1 standards. These features are provided through the `oracle.jdbc` and `oracle.sql` packages. These packages support Java Development Kit (JDK) releases 6 and 7. This chapter discusses the JDBC standards support in Oracle JDBC drivers. It contains the following sections:

Support for JDBC 2.0 Standard

JDK 1.2 and later versions provide support for JDBC 2.0 features. There are three areas to consider:

- Support for data types, such as objects, arrays, and large objects (LOBs), which is handled through the `java.sql` package.
- Support for standard features, such as result set enhancements and update batching, which is handled through standard objects, such as `Connection`, `ResultSet`, and `PreparedStatement`, under JDK 1.2.x and later.
- Support for extended features, such as features of the JDBC 2.0 optional package, also known as the standard extension application programming interface (API), including data sources, connection pooling, and distributed transactions.

This section covers the following topics:

Note: Versions of JDK earlier than 5.0 are no longer supported. The package `oracle.jdbc2` has been removed.

Data Type Support

Oracle JDBC fully supports JDK 6 and JDK 7, which includes standard JDBC 2.0 functionality through implementation of interfaces in the standard `java.sql` package. These interfaces are implemented as appropriate by classes in the `oracle.sql` and `oracle.jdbc` packages.

Standard Feature Support

In a JDK 6.0 environment, using the JDBC classes in `ojdbc6.jar`, JDBC 2.0 features, such as scrollable result sets, updatable result sets, and update batching, are supported through methods specified by standard JDBC 2.0 interfaces.

Extended Feature Support

Features of the JDBC 2.0 optional package, including data sources, connection pooling, and distributed transactions, are supported in a JDK 1.2.x or later environment.

The standard `javax.sql` package and classes that implement its interfaces are included in the Java Archive (JAR) files packaged with Oracle Database.

Standard versus Oracle Performance Enhancement APIs

The following performance enhancements are available under JDBC 2.0, which had previously been available only as Oracle extensions:

Note: Starting from Oracle Database 12*c* Release 1 (12.1), Oracle update batching is deprecated. Oracle recommends that you use standard JDBC batching instead of Oracle update batching.

- Update batching
- Fetch size or row prefetching

In each case, you have the option of using the standard model or the Oracle model. Oracle recommends that you use the JDBC standard model whenever possible. Do not, however, try to mix usage of the standard model and Oracle model within a single application for either of these features.

See Also:

- "Update Batching" on page 21-1
- "Fetch Size" on page 17-4

Support for JDBC 3.0 Standard

Standard JDBC 3.0 features are supported by JDK 1.4 and later versions. Table 3–1 lists the JDBC 3.0 features supported by Oracle Database 12*c* Release 1 (12.1) and gives references to a detailed discussion of each feature.

Table 3–1 Key Areas of JDBC 3.0 Functionality

Feature	Comments and References
Transaction savepoints	See "Transaction Savepoints" on page 3-3 for information.
Statement caching	Reuse of prepared statements by connection pools. See Chapter 20, "Statement and Result Set Caching".
Switching between local and global transactions	See "Switching Between Global and Local Transactions" on page 30-3.
LOB modification	See "JDBC 3.0 LOB Interface Methods" on page 3-5.
Named SQL parameters	See "Interface oracle.jdbc.OracleCallableStatement" on page 4-22 and "Interface oracle.jdbc.OraclePreparedStatement" on page 4-21.
RowSets	See Chapter 18, "JDBC RowSets"
Retrieving auto-generated keys	See "Retrieval of Auto-Generated Keys" on page 3-4
Result set holdability	See "Result Set Holdability" on page 3-6

The following JDBC 3.0 features supported by Oracle JDBC drivers are covered in this section:

- Transaction Savepoints
- Retrieval of Auto-Generated Keys
- JDBC 3.0 LOB Interface Methods
- Result Set Holdability

Transaction Savepoints

The JDBC 3.0 specification supports **savepoints**, which offer finer demarcation within transactions. Applications can set a savepoint within a transaction and then roll back all work done after the savepoint. Savepoints relax the atomicity property of transactions. A transaction with a savepoint is atomic in the sense that it appears to be a single unit outside the context of the transaction, but code operating within the transaction can preserve partial states.

Note: Savepoints are supported for local transactions only. Specifying a savepoint within a global transaction causes a `SQLException` exception to be thrown.

Creating a Savepoint

You create a savepoint using the `Connection.setSavepoint` method, which returns a `java.sql.Savepoint` instance.

A savepoint is either named or unnamed. You specify the name of a savepoint by supplying a string to the `setSavepoint` method. If you do not specify a name, then the savepoint is assigned an integer ID. You retrieve a name using the `getSavepointName` method. You retrieve an ID using the `getSavepointId` method.

Note: Attempting to retrieve a name from an unnamed savepoint or attempting to retrieve an ID from a named savepoint throws a `SQLException` exception.

Rolling Back to a Savepoint

You roll back to a savepoint using the `Connection.rollback(Savepoint svpt)` method. If you try to roll back to a savepoint that has been released, then a `SQLException` exception is thrown.

Releasing a Savepoint

You remove a savepoint using the `Connection.releaseSavepoint(Savepoint svpt)` method.

Checking Savepoint Support

You query if savepoints are supported by your database by calling the `oracle.jdbc.OracleDatabaseMetaData.supportsSavepoints` method, which returns `true` if savepoints are available, `false` otherwise.

Savepoint Notes

When using savepoints, you must consider the following:

- After a savepoint has been released, attempting to reference it in a rollback operation will cause a `SQLException` exception to be thrown.
- When a transaction is committed or rolled back, all savepoints created in that transaction are automatically released and become invalid.
- Rolling a transaction back to a savepoint automatically releases and makes invalid any savepoints created after the savepoint in question.

Retrieval of Auto-Generated Keys

Many database systems automatically generate a unique key field when a row is inserted. Oracle Database provides the same functionality with the help of sequences and triggers. JDBC 3.0 introduces the retrieval of auto-generated keys feature that enables you to retrieve such generated values. In JDBC 3.0, the following interfaces are enhanced to support the retrieval of auto-generated keys feature:

- `java.sql.DatabaseMetaData`
- `java.sql.Connection`
- `java.sql.Statement`

These interfaces provide methods that support retrieval of auto-generated keys. However, this feature is supported only when `INSERT` statements are processed. Other data manipulation language (DML) statements are processed, but without retrieving auto-generated keys.

Note: The Oracle server-side internal driver does not support the retrieval of auto-generated keys feature.

---

java.sql.Statement

If key columns are not explicitly indicated, then Oracle JDBC drivers cannot identify which columns need to be retrieved. When a column name or column index array is used, Oracle JDBC drivers can identify which columns contain auto-generated keys that you want to retrieve. However, when the `Statement.RETURN_GENERATED_KEYS` integer flag is used, Oracle JDBC drivers cannot identify these columns. When the integer flag is used to indicate that auto-generated keys are to be returned, the `ROWID`

pseudo column is returned as key. The ROWID can be then fetched from the ResultSet object and can be used to retrieve other columns.

Sample Code

The following code illustrates retrieval of auto-generated keys:

```
/** SQL statements for creating an ORDERS table and a sequence for generating the
  * ORDER_ID.
  *
  * CREATE TABLE ORDERS (ORDER_ID NUMBER, CUSTOMER_ID NUMBER, ISBN NUMBER,
  * DESCRIPTION NCHAR(5))
  *
  * CREATE SEQUENCE SEQ01 INCREMENT BY 1 START WITH 1000
  */

...
String cols[] = {"ORDER_ID", "DESCRIPTION"};

// Create a PreparedStatement for inserting a row into the ORDERS table.
OraclePreparedStatement pstmt = (OraclePreparedStatement)
conn.prepareStatement("INSERT INTO ORDERS (ORDER_ID, CUSTOMER_ID, ISBN,
DESCRIPTION) VALUES (SEQ01.NEXTVAL, 101,
 966431502, ?)", cols);
char c[] = {'a', '\u5185', 'b'};
String s = new String(c);
pstmt.setNString(1, s);
pstmt.executeUpdate();
ResultSet rset = pstmt.getGeneratedKeys();
...
```

In the preceding example, a sequence, SEQ01, is created to generate values for the ORDER_ID column starting from 1000 and incrementing by 1 each time the sequence is processed to generate the next value. An OraclePreparedStatement object is created to insert a row in to the ORDERS table.

Limitations

Auto-generated keys are implemented using the DML returning clause. So, they are subjected to the following limitations:

- You cannot combine auto-generated keys with batch update.
- You need to access the ResultSet object returned from getGeneratedKeys method by position only and no bind variable names should be used as columns in the ResultSet object.

JDBC 3.0 LOB Interface Methods

Table 3–2 and Table 3–3 show the conversions between Oracle proprietary methods and JDBC 3.0 standard methods.

Table 3–2 BLOB Method Equivalents

Oracle Proprietary Method	JDBC 3.0 Standard Method
putBytes(long pos, byte [] bytes)	setBytes(long pos, byte[] bytes)
putBytes(long pos, byte [] bytes, int length)	setBytes(long pos, byte[] bytes, int offset, int len)
getBinaryOutputStream(long pos)	setBinaryStream(long pos)

Table 3–2 (Cont.) BLOB Method Equivalents

Oracle Proprietary Method	JDBC 3.0 Standard Method
`trim (long len)`	`truncate(long len)`

Table 3–3 CLOB Method Equivalents

Oracle Proprietary Method	JDBC 3.0 Standard Method
`putString(long pos, String str)`	`setString(long pos, String str)`
not applicable	`setString(long pos, String str, int offset, int len)`
`getAsciiOutputStream(long pos)`	`setAsciiStream(long pos)`
`getCharacterOutputStream(long pos)`	`setCharacterStream(long pos)`
`trim (long len)`	`truncate(long len)`

Result Set Holdability

Result set holdability was introduced since JDBC 3.0. This feature enables applications to decide whether the `ResultSet` objects should be open or closed, when a commit operation is performed. The commit operation could be either implicit or explicit.

Oracle Database supports only `HOLD_CURSORS_OVER_COMMIT`. Therefore, it is the default value for Oracle JDBC drivers. Any attempt to change holdability will throw a `SQLFeatureNotSupportedException` exception.

Support for JDBC 4.0 Standard

The JDBC 4.0 standard support is provided by JDK 6 and later versions. Oracle Database 12*c* Release 1 (12.1) JDBC drivers provide support for the JDBC 4.0 standard.

Note:

- You need to have the `ojdbc6.jar` and `ojdbc7.jar` in your `classpath` environment variable in order to have JDBC 4.0 standard support with JDK 6 and JDK 7 respectively.
- The JDBC 4.0 specification defines the `java.sql.Connection.createArrayOf` factory method to create `java.sql.Array` objects. The `createArrayOf` method accepts the name of the array element type as one of the arguments, where the array type is anonymous. Oracle database supports only named array types, not anonymous array types. So, the current release of Oracle JDBC drivers do not and cannot support the `createArrayOf` method. You must use the Oracle specific `createARRAY` method to create an array type. For more information about the `createArrayOf` method, refer to "Creating ARRAY Objects" on page 16-5.
- This document provides only an overview of these new features. For detailed information about these features, see *"Java 2 Platform, Standard Edition (JSE) 6.0 specification*" at

  `http://docs.oracle.com/javase/6/docs/`

Some of the features available in Oracle Database 12*c* Release 1 (12.1) JDBC drivers are the following:

- Wrapper Pattern Support
- SQLXML Type
- Enhanced Exception Hierarchy and SQLException
- The RowId Data Type
- LOB Creation
- National Language Character Set Support

Wrapper Pattern Support

Wrapper pattern is a common coding pattern used in Java applications to provide extensions beyond the traditional JDBC API that are specific to a data source. You may need to use these extensions to access the resources that are wrapped as proxy class instances representing the actual resources. JDBC 4.0 introduces the `Wrapper` interface that describes a standard mechanism to access these wrapped resources represented by their proxy, to permit direct access to the resource delegates.

The `Wrapper` interface provides the following two methods:

- `public boolean isWrapperFor(Class<?> iface) throws SQLException;`
- `public <T> T unwrap(Class<T> iface) throws SQLException;`

The other JDBC 4.0 interfaces, except those that represent SQL data, all implement this interface. These include `Connection`, `Statement` and its subtypes, `ResultSet`, and the metadata interfaces.

See Also:

`http://docs.oracle.com/javase/7/docs/api/java/sql/Wrapper.html`

SQLXML Type

One of the most important updates in JDBC 4.0 standard is the support for the XML data type, defined by the SQL 2003 standard. Now JDBC offers a mapping interface to support the SQL/XML database data type, that is, `java.sql.SQLXML`. This new JDBC interface defines Java native bindings for XML, thus making handling of any database XML data easier and more efficient.

Note:

- You also need to include the `xdb6.jar` and `xmlparserv2.jar` files in the `classpath` environment variable to use `SQLXML` type data, if they are not already present in the `classpath`.
- SQLXML is not supported in `CachedRowset` objects.

You can create an instance of XML by calling the `createSQLXML` method in `java.sql.Connection` interface. This method returns an empty XML object.

The `PreparedStatement`, `CallableStatement`, and `ResultSet` interfaces have been extended with the appropriate getter and setter methods in the following way:

- `PreparedStatement`: The method `setSQLXML` have been added

- `CallableStatement`: The methods `getSQLXML` and `setSQLXML` have been added
- `ResultSet`: The method `getSQLXML` have been added

> **Note:** In Oracle Database 10*g* and earlier versions of Oracle Database 11*g*, Oracle JDBC drivers supported the Oracle SQL XML type (XMLType) through an Oracle proprietary extension, which did not conform to the JDBC standard.
>
> The 11.2.0.2 Oracle JDBC drivers conformed to the JDBC standard with the introduction of a new connection property, `oracle.jdbc.getObjectReturnsXMLType`. If you set this property to `false`, then the `getObject` method returns an instance of `java.sql.SQLXML` type and if you depend on the existing Oracle proprietary support for SQL XMLType using `oracle.xdb.XMLType`, then you can change the value of this property back to `true`.
>
> However, setting of the `getObjectReturnsXMLType` property is not required for the current version of Oracle JDBC drivers.

Example

Example 3–1 Accessing SQLXML Data

The following example shows how to create an instance of XML from a `String`, write the XML data into the Database, and then retrieve the XML data from the Database.

```
import java.sql.*;
import java.util.Properties;
import oracle.jdbc.pool.OracleDataSource;

public class SQLXMLTest
 {

  public static void main(String[] args)
  {

  Connection conn = null;
  Statement stmt = null;
  ResultSet rs = null;
  PreparedStatement ps = null;

  String xml = "<?xml version=\"1.0\"?>\n" +
    "<oldjoke>\n" +
    "<burns>Say <quote>goodnight</quote>, Gracie.</burns>\n" +
    "<allen><quote>Goodnight, Gracie.</quote></allen>\n" +
    "<applause/>\n" +
    "</oldjoke>";

  try
  {

     OracleDataSource ods = new OracleDataSource();
     ods.setURL("jdbc:oracle:thin:@//localhost:5221/orcl");
     ods.setUser("HR");
     ods.setPassword("hr");
     conn = ods.getConnection();

     ps = conn.prepareStatement("insert into x values (?, ?)");
```

```
    ps.setString(1, "string to string");
    SQLXML x = conn.createSQLXML();
    x.setString(xml);
    ps.setSQLXML(2, x);
    ps.execute();
    stmt = conn.createStatement();
    rs = stmt.executeQuery("select * from x");
    while (rs.next())
    {
      x = rs.getSQLXML(2);
      System.out.println(rs.getString(1) + "\n" + rs.getSQLXML(2).getString());
      x.free();
    }

    rs.close();
    ps.close();
  }

  catch (SQLException e){e.printStackTrace ();}

  }
}
```

Note: Calling a setter method with an empty XML throws `SQLException`. The getter methods never return an empty XML.

See Also: JSR 173: Streaming API for XML at:

http://www.jcp.org/aboutJava/communityprocess/first/jsr173/

Enhanced Exception Hierarchy and SQLException

JDBC 3.0 defines only a single exception, `SQLException`. However, there are large categories of errors and it is useful to distinguish them. This feature provides subclasses of the `SQLException` class to identify the different categories of errors. The primary distinction is between permanent errors and transient errors. Permanent errors are a result of the correct operation of the system and will always occur. Transient errors are the result of failures, including timeouts, of some part of the system and may not reoccur.

JDBC 4.0 adds additional exceptions to represent transient and permanent errors and the different categories of these errors.

Also, the `SQLException` class and its subclasses are enhanced to provide support for the J2SE chained exception functionality.

The RowId Data Type

JDBC 4.0 provides the `java.sql.RowId` data type to represent SQL `ROWID` values. You can retrieve a `RowId` value using the getter methods defined in the `ResultSet` and `CallableStatement` interfaces. You can also use a `RowId` value in a parameterized `PreparedStatement` to set a parameter with a `RowId` object or in an updatable result set to update a column with a specific `RowId` value.

A `RowId` object is valid until the identified row is not deleted. A `RowId` object may also be valid for the following:

- The duration of the transaction in which it is created
- The duration of the session in which it is created
- An undefined duration where by it is valid forever

The lifetime of the RowId object can be determined by calling the `DatabaseMetaData.getRowIdLifetime` method.

LOB Creation

In JDBC 4.0, the `Connection` interface has been enhanced to provide support for the creation of `BLOB`, `CLOB`, and `NCLOB` objects. The interface provides the `createBlob`, `createClob`, and `createNClob` methods that enable you to create `Blob`, `Clob`, and `NClob` objects.

The created large objects (LOBs) do not contain any data. You can add or retrieve data to or from these objects by calling the APIs available in the `java.sql.Blob`, `java.sql.Clob`, and `java.sql.NClob` interfaces. You can either retrieve the entire content or a part of the content from these objects. The following code snippet illustrates how to retrieve 100 bytes of data from a `BLOB` object starting at offset 200:

```
...
Connection con = DriverManager.getConnection(url, props);
Blob aBlob = con.createBlob();
// Add data to the BLOB object.
aBlob.setBytes(...);
...
// Retrieve part of the data from the BLOB object.
InputStream is = aBlob.getBinaryStream(200, 100);
...
```

You can also pass LOBs as input parameters to a `PreparedStatement` object by using the `setBlob`, `setClob`, and `setNClob` methods. You can use the `updateBlob`, `updateClob`, and `updateNClob` methods to update a column value in an updatable result set.

These LOBs are temporary LOBs and can be used for any purpose for which temporary LOBs should be used. To make the storage permanent in the database, these LOBs must be written to a table.

See Also: "Working With Temporary LOBs" on page 14-8

Temporary LOBs remain valid for at least the duration of the transaction in which they are created. This may result in unwarranted use of memory during a long running transaction. You can release LOBs by calling their `free` method, as follows:

```
...
Clob aClob = con.createClob();
int numWritten = aClob.setString(1, val);
aClob.free();
...
```

National Language Character Set Support

JDBC 4.0 introduces the `NCHAR`, `NVARCHAR`, `LONGNVARCHAR`, and `NCLOB` JDBC types to access the national character set types. These types are similar to the `CHAR`, `VARCHAR`, `LONGVARCHAR`, and `CLOB` types, except that the values are encoded using the national character set.

Support for JDBC 4.1 Standard

Oracle Database 12*c* Release 1 (12.1) JDBC drivers provide support for JDBC 4.1 standard through JDK 7. This section describes the following methods that are supported in this release:

- setClientInfo Method
- getObject Method

setClientInfo Method

The `setClientInfo` method sets the value of the property providing client information. This method accepts keys of the form `<namespace>.<keyname>`. The JDBC driver supports any `<namespace>.<keyname>` combination.

Note: The `setClientInfo` method supports the `OCSID` namespace among other namespaces. But, there are differences between using the `OCSID` namespace and any other namespace. With the `OCSID` namespace, the `setClientInfo` method supports only the following keys:

- `ACTION`
- `CLIENTID`
- `ECID`
- `MODULE`
- `SEQUENCE_NUMBER`
- `DBOP`

Also, the information associated with any other namespace is communicated through the network using a single protocol, while information associated with the OCSID namespace is communicated using a different protocol. The protocol used for the OCSID namespace is also used by the OCI C Library and the 10*g* thin driver to send end-to-end metrics values.

The `setClientInfo` method checks the Java permission `oracle.jdbc.clientInfo` and if the security check fails, then it throws a `SecurityException`. It supports permission name patterns of the form `<namespace>.*`. The `setClientInfo` method either sets or clears all pairs, so it requires that the permission name must be set to an asterisk (`*`).

The `setClientInfo` method is backward compatible with the `setEndToEndMetrics` and the `setClientIdentifier` methods, and can use DMS to set client tags.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), the `setEndToEndMetrics` method is deprecated.

See Also: Chapter 33, "JDBC DMS Metrics" for more information about end-to-end metrics and DMS metrics.

Monitoring Database Operations

Many Java applications do not have a database connection, but they need to track database activities on behalf of their functionalities. For such applications, Oracle

Database 12*c* Release 1 (12.1) introduces the `DBOP` tag that can be associated with a thread in the application when the application does not have explicit access to a database. The DBOP tag is associated with a thread through the invocation of DMS APIs, without requiring an active connection to the database. When the thread sends the next database call, then DMS propagates these tags through the connection along with the database call, without requiring an extra round trip. In this way, applications can associate their activity with database operations while factorizing the code in the Application layer.

The `DBOP` tag composes of the following:

- Database operation name
- The execution ID
- Operation attributes

The `setClientInfo` method supports the `DBOP` tag. The `setClientInfo` method sets the value of the tag to monitor the database operations. When the JDBC application connects to the database and a database round-trip is made, the database activities can be tracked. For example, you can set the value of the `DBOP` tag to `foo` in the following way:

```
...
Connection conn = DriverManager.getConnection(myUrl, myUsername, myPassword);
conn.setClientInfo("E2E_CONTEXT.DBOP", "foo");
Statement stmt = conn.createStatement();
stmt.execute("select 1 from dual");  // DBOP tag is set after this
...
```

getObject Method

The `getObject` method retrieves an object, based on the parameters passed. Oracle Database 12*c* Release 1 (12.1) supports the following two new `getObject` methods:

Method 1

```
<T> T getObject(int parameterIndex,
                java.lang.Class<T> type)
            throws SQLException
```

Method 2

```
<T> T getObject(java.lang.String parameterName,
                java.lang.Class<T> type)
            throws SQLException
```

These methods support the conversions listed in the JDBC specification and also the additional conversions listed in Table A–1. The Oracle Database 12*c* Release 1 (12.1) drivers also support conversions to some additional classes, which implement one or more static `valueOf` methods, if any of the following criteria is met:

- No other conversion is specified in JDBC specification or Table A–1
- The `type` argument defines one or more public static single argument methods named `valueOf`
- One or more of the `valueOf` methods take an argument that is a value of a type supported because of JDBC specification or Table A–1

This release of JDBC drivers convert the value to a type specified in the JDBC specification, or in Table A–1 and then call the corresponding `valueOf` method with the converted value as the argument. If there is more than one appropriate `valueOf` method, then the JDBC driver chooses one `valueOf` method in an unspecified way.

Example

```
ResultSet rs = . . . ;
Character c = rs.getObject(1, java.lang.Character.class);
```

The Character class defines the following `valueOf` method:

```
public static Character valueOf(char c);
```

Table A–1 specifies that `NUMBER` can be converted to `char`. So, if the first column of the `ResultSet` is a `NUMBER`, then the `getObject` method converts that `NUMBER` value to a `char` and passes the `char` value to the `valueOf(char)` method and returns the resulting `Character` object.

4

Oracle Extensions

Oracle provides Java classes and interfaces that extend the Java Database Connectivity (JDBC) standard implementation, enabling you to access and manipulate Oracle data types and use Oracle performance extensions. This chapter provides an overview of the classes and interfaces provided by Oracle that extend the JDBC standard implementation. It also describes some of the key support features of the extensions.

This chapter contains the following sections:

> **Note:** This chapter focuses on type extensions, as opposed to performance extensions, which are discussed in detail in Chapter 21, "Performance Extensions".

Overview of Oracle Extensions

Beyond standard features, Oracle JDBC drivers provide Oracle-specific type extensions and performance extensions. These extensions are provided through the following Java packages:

- `oracle.sql`

  Provides classes that represent SQL data in Oracle format

- `oracle.jdbc`

  Provides interfaces to support database access and updates in Oracle type formats

> **See Also:** "Oracle JDBC Packages" on page 4-5

Features of the Oracle Extensions

The Oracle extensions to JDBC include a number of features that enhance your ability to work with Oracle Databases. These include the following:

Database Management Using JDBC

Starting from Oracle Database 11*g* Release 1, the `oracle.jdbc.OracleConnection` interface has two JDBC methods, `startup` and `shutdown`, which enable you to start up and shut down an Oracle Database instance. You also have support for the Database Change Notification feature of Oracle Database. These new features are discussed in details in "Database Administration".

Note: My Oracle Support Note 335754.1 announces the desupport of the `oracle.jdbc.driver.*` package in Oracle Database 11*g* JDBC drivers. In other words, Oracle Database 10*g* Release 2 was the last database to support this package and any API depending on the `oracle.jdbc.driver.*` package will fail to compile in the current release of the Database. You must remove such APIs and migrate to the standard APIs. For example, if your code uses the `oracle.jdbc.CustomDatum` and `oracle.jdbc.CustomDatumFactory` interfaces, then you must replace them with the `java.sql.Struct` or `java.sql.SQLData` interfaces.

See Also: My Oracle Support Note 1364193.1, "New JDBC Interfaces for Oracle types" for more information

Support for Oracle Data Types

One of the features of the Oracle JDBC extensions is the type support in the `oracle.sql` package. This package includes classes that are an exact representation of the data in Oracle format. Keep the following important points in mind, when you use `oracle.sql` types in your program:

- For numeric type of data, the conversion to standard Java types does not guarantee to retain full precision due to limitations of the data conversion process. Use the `BigDecimal` type to minimize any data loss issues.
- For certain data types, the conversion to standard Java types can be dependent on the system settings and your program may not run as expected. This is a known limitation while converting data from `oracle.sql` types to standard Java types.
- If the functionalities of your program is limited to reading data from one table and writing the same to another table, then for numeric and date data, `oracle.sql` types are slightly faster as compared to standard Java types. But, if your program involves even a simple data manipulation operation like compare or print, then standard Java types are faster.
- `oracle.sql.CHAR` is not an exact representation of the data in Oracle format. `oracle.sql.CHAR` is constructed from `java.lang.String`. There is no advantage of using `oracle.sql.CHAR` because `java.lang.String` is always faster and represents the same character sets, excluding a couple of desupported character sets.

Note: Oracle strongly recommends you to use standard Java types and convert any existing `oracle.sql` type of data to standard Java types. Internally, the Oracle JDBC drivers strive to maximize the performance of Java standard types. `oracle.sql` types are supported *only* for backward compatibility and their use is discouraged.

See Also:

- Package oracle.sql on page 4-5
- "Oracle Character Data Types Support" on page 4-9
- "Additional Oracle Type Extensions" on page 4-12

Support for Oracle Objects

Oracle JDBC supports the use of structured objects in the database, where an object data type is a user-defined type with nested attributes. For example, a user application could define an `Employee` object type, where each `Employee` object has a `firstname` attribute (character string), a `lastname` attribute (character string), and an `employeenumber` attribute (integer).

Oracle JDBC supports Oracle object data types. When you work with Oracle object data types in a Java application, you must consider the following:

- How to map between Oracle object data types and Java classes
- How to store Oracle object attributes in corresponding Java objects
- How to convert attribute data between SQL and Java formats
- How to access data

Oracle objects can be mapped either to the weak `java.sql.Struct` type or to strongly typed customized classes. These strong types are referred to as custom Java classes, which must implement either the standard `java.sql.SQLData` interface or the Oracle extension `oracle.jdbc.OracleData` interface. Each interface specifies methods to convert data between SQL and Java.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), the `OracleData` interface has replaced the `ORAData` interface.

Oracle recommends the use of the Oracle JPublisher utility to create custom Java classes to correspond to your Oracle objects. Oracle JPublisher performs this task seamlessly with command-line options and can generate either `SQLData` or `OracleData` interface implementations.

For `SQLData` interface implementations, a type map defines the correspondence between Oracle object data types and Java classes. **Type maps** are objects that specify which Java class corresponds to each Oracle object data type. Oracle JDBC uses these type maps to determine which Java class to instantiate and populate when it retrieves Oracle object data from a result set.

Note: Oracle recommends using the `OracleData` interface, instead of the `SQLData` interface, in situations where portability is not a concern. The `OracleData` interface works more easily and flexibly in conjunction with other features of the Oracle platform offerings using Java.

JPublisher automatically defines `get`*`XXX`* methods of the custom Java classes, which retrieve data into your Java application.

See Also:

- Chapter 13, "Working with Oracle Object Types"
- *Oracle Database JPublisher User's Guide*

Support for Schema Naming

Oracle object data type classes have the ability to accept and return fully qualified schema names. A fully qualified schema name has this syntax:

```
{[schema_name].}[sql_type_name]
```

Where, *`schema_name`* is the name of the schema and *`sql_type_name`* is the SQL type name of the object. *`schema_name`* and *`sql_type_name`* are separated by a period (`.`).

To specify an object type in JDBC, use its fully qualified name. It is not necessary to enter a schema name if the type name is in the current naming space, that is, the current schema. Schema naming follows these rules:

- Both the schema name and the type name may or may not be within quotation marks. However, if the SQL type name has a period in it, such as `CORPORATE.EMPLOYEE`, the type name must be quoted.
- The JDBC driver looks for the first period in the object name that is not within quotation marks and uses the string before the period as the schema name and the string following the period as the type name. If no period is found, then the JDBC driver takes the current schema as default. That is, you can specify only the type name, without indicating a schema, instead of specifying the fully qualified name if the object type name belongs to the current schema. This also explains why you must put the type name within quotation marks if the type name has a dot in it.

  For example, assume that user `HR` creates a type called `person.address` and then wants to use it in his session. `HR` may want to skip the schema name and pass in `person.address` to the JDBC driver. In this case, if `person.address` is not within quotation marks, then the period is detected and the JDBC driver mistakenly interprets `person` as the schema name and `address` as the type name.
- JDBC passes the object type name string to the database unchanged. That is, the JDBC driver does not change the character case even if the object type name is within quotation marks.

  For example, if `HR.PersonType` is passed to the JDBC driver as an object type name, then the JDBC driver passes the string to the database unchanged. As another example, if there is white space between characters in the type name string, then the JDBC driver will not remove the white space.

DML Returning

Oracle Database supports the use of the `RETURNING` clause with data manipulation language (DML) statements. This enables you to combine two SQL statements into one. Both the Oracle JDBC Oracle Call Interface (OCI) driver and the Oracle JDBC Thin driver support DML returning.

See Also: "DML Returning" on page 4-24

Accessing PL/SQL Associative Arrays

Oracle JDBC drivers enable JDBC applications to make PL/SQL calls with Associative Array parameters. Oracle JDBC drivers support PL/SQL Associative Arrays of scalar data types

See Also: "Accessing PL/SQL Associative Arrays" on page 4-27

Oracle JDBC Packages

This section describes the following Java packages, which support the Oracle JDBC extensions:

- Package oracle.sql
- Package oracle.jdbc

Package oracle.sql

The `oracle.sql` package supports direct access to data in SQL format. This package consists primarily of classes that provide Java mappings to SQL data types and their support classes. Essentially, the classes act as Java containers for SQL data.

Each of the `oracle.sql.*` data type classes extends `oracle.sql.Datum`, a superclass that encapsulates functionality common to all the data types. Some of the classes are for JDBC 2.0-compliant data types. These classes, implement standard JDBC 2.0 interfaces in the `java.sql` package, as well as extending the `oracle.sql.Datum` class.

The `LONG` and `LONG RAW` SQL types and `REF CURSOR` type category have no `oracle.sql.*` classes. Use standard JDBC functionality for these types. For example, retrieve `LONG` or `LONG RAW` data as input streams using the standard JDBC result set and callable statement methods `getBinaryStream` and `getCharacterStream`. Use the `getCursor` method for `REF CURSOR` types.

Note: Oracle recommends the use of standard JDBC types or Java types whenever possible. The types in the package `oracle.sql.*` are provided primarily for backward compatibility or for support of a few Oracle specific features such as `OPAQUE`, `OracleData`, `TIMESTAMPTZ`, and so on.

General oracle.sql.* Data Type Support

Each of the Oracle data type classes provides, among other things, the following:

- Data storage as Java byte arrays for SQL data
- A `getBytes()` method, which returns the SQL data as a byte array
- A `toJdbc()` method that converts the data into an object of a corresponding Java class as defined in the JDBC specification

The JDBC driver does not convert Oracle-specific data types that are not part of the JDBC specification, such as `BFILE`. The driver returns the object in the corresponding `oracle.sql.*` format.

- Appropriate *xxx*`Value` methods to convert SQL data to Java type. For example, `stringValue`, `intValue`, `booleanValue`, `dateValue`, and `bigDecimalValue`
- Additional conversion methods, `get`*`XXX`* and `set`*`XXX`*, as appropriate, for the functionality of the data type, such as methods in the large object (LOB) classes that get the data as a stream and methods in the `REF` class that get and set object data through the object reference.

Overview of Class oracle.sql.STRUCT

`oracle.sql.STRUCT` class is the Oracle implementation of `java.sql.Struct` interface. This class is a value class and you should not change the contents of the class after construction. This class, as with all `oracle.sql.*` data type classes, is a subclass of the `oracle.sql.Datum` class.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.STRUCT` class is deprecated and replaced with the `oracle.jdbc.OracleStruct` interface, which is a part of the `oracle.jdbc` package. Oracle strongly recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleStruct` interface.

See Also: *Oracle Database JDBC Java API Reference*

Overview of Class oracle.sql.REF

The `oracle.sql.REF` class is the generic class that supports Oracle object references. This class, as with all `oracle.sql.*` data type classes, is a subclass of the `oracle.sql.Datum` class.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.REF` class is deprecated and replaced with the `oracle.jdbc.OracleRef` interface, which is a part of the `oracle.jdbc` package. Oracle strongly recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleRef` interface.

The `REF` class has methods to retrieve and pass object references. However, selecting an object reference retrieves only a pointer to an object. This does not materialize the object itself. But the `REF` class also includes methods to retrieve and pass the object data. You cannot create `REF` objects in your JDBC application. You can only retrieve existing `REF` objects from the database.

You should use the JDBC standard type, `java.sql.Ref`, and the JDBC standard methods in preference to using `oracle.sql.REF`. If you want your code to be more portable, then you must use the standard type because only the Oracle JDBC drivers will use instances of `oracle.sql.REF` type.

See Also: *Oracle Database JDBC Java API Reference*

Overview of Classes oracle.sql.BLOB, oracle.sql.CLOB, oracle.sql.BFILE

Binary large objects (BLOBs), character large objects (CLOBs), and binary files (BFILEs) are for data items that are too large to store directly in a database table. Instead, the database table stores a locator that points to the location of the actual data.

Note:

- Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.BLOB` and `Oracle.sql.CLOB` classes are deprecated and replaced with the `oracle.jdbc.OracleBlob` and `oracle.jdbc.OracleClob` interfaces respectively, which are a part of the `oracle.jdbc` package. Oracle strongly recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleBlob` and `oracle.jdbc.OracleClob` interfaces.
- `oracle.sql.BFILE` is an Oracle proprietary extension and there is no JDBC standard equivalent.

The `oracle.sql` package supports these data types in several ways:

- BLOBs point to large unstructured binary data items and are supported by the `oracle.sql.BLOB` class.
- CLOBs point to large character data items and are supported by the `oracle.sql.CLOB` class.
- BFILEs point to the content of external files (operating system files) and are supported by the `oracle.sql.BFILE` class. BFiles are read-only.

You can select a BLOB, CLOB, or BFILE locator from the database using a standard `SELECT` statement. However, you receive only the locator, and not the data. Additional steps are necessary to retrieve the data.

See Also: Chapter 14, "Working with LOBs and BFILEs".

Overview of Classes oracle.sql.DATE, oracle.sql.NUMBER, and oracle.sql.RAW

These classes hold primitive SQL data types in Oracle native representation. In most cases, these types are not used internally by the drivers and you should use the standard JDBC types instead.

Java `Double` and `Float` `NaN` values do not have an equivalent Oracle `NUMBER` representation. For example, for Oracle `BINARY_FLOAT` and `BINARY_DOUBLE` data types, negative zero is coerced to positive zero and all NaNs are coerced to the canonical one. So, a `NullPointerException` is thrown whenever a `Double.NaN` value or a `Float.NaN` value is converted into an Oracle `NUMBER` using the `oracle.sql.NUMBER` class. For instance, the following code throws a `NullPointerException`:

```
oracle.sql.NUMBER n = new oracle.sql.NUMBER(Double.NaN);
System.out.println(n.doubleValue());  // throws NullPointerException
```

See Also: *Oracle Database SQL Language Reference*

Overview of Classes oracle.sql.TIMESTAMP, oracle.sql.TIMESTAMPTZ, and oracle.sql.TIMESTAMPLTZ

The JDBC drivers support the following date/time data types:

- `TIMESTAMP` (`TIMESTAMP`)
- `TIMESTAMP WITH TIME ZONE` (`TIMESTAMPTZ`)
- `TIMESTAMP WITH LOCAL TIME ZONE` (`TIMESTAMPLTZ`)

The JDBC drivers allow conversions between `DATE` and date/time data types. For example, you can access a `TIMESTAMP WITH TIME ZONE` column as a `DATE` value.

The JDBC drivers support the most popular time zone names used in the industry as well as most of the time zone names defined in the JDK. Time zones are specified by using the `java.util.TimeZone` class.

Note:

- Do not use `TimeZone.getTimeZone` to create time zone objects. The Oracle time zone data types support more time zone names than JDK.
- If a result set contains a `TIMESTAMPLTZ` column followed by a `LONG` column, then reading the `LONG` column results in an error.

The following code shows how the `TimeZone` and `Calendar` objects are created for `US_PACIFIC`, which is a time zone name not defined in JDK:

```
TimeZone tz = TimeZone.getDefault();
tz.setID("US_PACIFIC");
GregorianCalendar gcal = new GregorianCalendar(tz);
```

The following Java classes represent the SQL date/time types:

- `oracle.sql.TIMESTAMP`
- `oracle.sql.TIMESTAMPTZ`
- `oracle.sql.TIMESTAMPLTZ`

Before accessing `TIMESTAMP WITH LOCAL TIME ZONE` data, call the `OracleConnection.setSessionTimeZone(String regionName)` method to set the session time zone. When this method is called, the JDBC driver sets the session time zone of the connection and saves the session time zone so that any `TIMESTAMP WITH LOCAL TIME ZONE` data accessed through JDBC can be adjusted using the session time zone.

Note: `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` types can be represented as standard `java.sql.Timestamp` type. The byte representation of `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` types to `java.sql.Timestamp` is straight forward. This is because the internal format of `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` data types is GMT, and `java.sql.Timestamp` type objects internally use a milliseconds time value that is the number of milliseconds since EPOCH. However, the `String` representation of these data types requires time zone information that is obtained dynamically from the server and cached on the client side.

In earlier versions of JDBC drivers, the cache of time zone was shared across different connections. This used to cause problems sometimes due to incompatibility in various time zones. Starting from Oracle Database 11 Release 2 version of JDBC drivers, the time zone cache is based on the time zone version supplied by the database. This newly designed cache avoids any issues related to version incompatibility of time zones.

Overview of Class oracle.sql.OPAQUE

The `oracle.sql.OPAQUE` class provides the name and characteristics of the `OPAQUE` type and any attributes. The `OPAQUE` type provides access only to the uninterrupted bytes of the instance.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.OPAQUE` class is deprecated and replaced with the `oracle.jdbc.OracleOpaque` interface, which is a part of the `oracle.jdbc` package. Oracle recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleOpaque` interface.

Package oracle.jdbc

The interfaces of the `oracle.jdbc` package define the Oracle extensions to the interfaces in `java.sql`. These extensions provide access to Oracle SQL-format data and other Oracle-specific functionality, including Oracle performance enhancements.

See Also: "The oracle.jdbc Package" on page 4-18

Oracle Character Data Types Support

Oracle character data types include the SQL CHAR and NCHAR data types. The following sections describe how these data types can be accessed using the `oracle.sql.*` classes:

SQL CHAR Data Types

The SQL CHAR data types include CHAR, VARCHAR2, and CLOB. These data types let you store character data in the database character set encoding scheme. The character set of the database is established when you create the database.

SQL NCHAR Data Types

The SQL NCHAR data types were created for Globalization Support. The SQL NCHAR data types include NCHAR, NVARCHAR2, and NCLOB. These data types enable you to store Unicode data in the database NCHAR character set encoding. The NCHAR character set, which never changes, is established when you create the database.

> **Note:** Because the UnicodeStream class is deprecated in favor of the CharacterStream class, the setUnicodeStream and getUnicodeStream methods are not supported for NCHAR data type access. Use the setCharacterStream method and the getCharacterStream method if you want to use stream access.

The usage of SQL NCHAR data types is similar to that of the SQL CHAR data types. JDBC uses the same classes and methods to access SQL NCHAR data types that are used for the corresponding SQL CHAR data types. Therefore, there are no separate, corresponding classes defined in the oracle.sql package for SQL NCHAR data types. Similarly, there is no separate, corresponding constant defined in the oracle.jdbc.OracleTypes class for SQL NCHAR data types.

The following code shows how to access SQL NCHAR data:

```
//
// Table TEST has the following columns:
// - NUMBER
// - NVARCHAR2
// - NCHAR
//
oracle.jdbc.OraclePreparedStatement pstmt =
  (oracle.jdbc.OraclePreparedStatement)
conn.prepareStatement("insert into TEST values(?, ?, ?)");

//
// oracle.jdbc.OraclePreparedStatement.FORM_NCHAR should be used for all NCHAR,
// NVARCHAR2 and NCLOB data types.
//

pstmt.setInt(1, 1);                    // NUMBER column
pstmt.setNString(2, myUnicodeString1);  // NVARCHAR2 column
pstmt.setNString(3, myUnicodeString2);  // NCHAR column
pstmt.execute();
```

Class oracle.sql.CHAR

The oracle.sql.CHAR class is used by Oracle JDBC in handling and converting character data. This class provides the Globalization Support functionality to convert character data. This class has two key attributes: Globalization Support character set and the character data. The Globalization Support character set defines the encoding of the character data. It is a parameter that is always passed when a CHAR object is constructed. Without the Globalization Support character set information, the data

bytes in the CHAR object are meaningless. The oracle.sql.CHAR class is used for both SQL CHAR and SQL NCHAR data types.

> **Note:** In versions of Oracle JDBC drivers prior to 10*g* Release 1, there were performance advantages to using the oracle.SQL.CHAR. Starting from Oracle Database 10*g*, there are no longer any such advantages. In fact, optimum performance is achieved using the java.lang.String. All Oracle JDBC drivers handle all character data in the Java UCS2 character set. Using the oracle.sql.CHAR does not prevent conversions between the database character set and UCS2 character set.

The only remaining use of the oracle.sql.CHAR class is to handle character data in the form of raw bytes encoded in an Oracle Globalization Support character set. All character data retrieved from Oracle Database should be accessed using the java.lang.String class. When processing byte data from another source, you can use an oracle.sql.CHAR to convert the bytes to java.lang.String.

To convert an oracle.sql.CHAR, you must provide the data bytes and an oracle.sql.CharacterSet instance that represents the Globalization Support character set used to encode the data bytes.

The CHAR objects that are Oracle object attributes are returned in the database character set.

JDBC application code rarely needs to construct CHAR objects directly, because the JDBC driver automatically creates CHAR objects, when it is needed to create them on those rare occasions.

To construct a CHAR object, you must provide character set information to the CHAR object by way of an instance of the CharacterSet class. Each instance of this class represents one of the Globalization Support character sets that Oracle supports. A CharacterSet instance encapsulates methods and attributes of the character set, mainly involving functionality to convert to or from other character sets.

Constructing an oracle.sql.CHAR Object

Follow these general steps to construct a CHAR object:

1. Create a CharacterSet object by calling the static CharacterSet.make method.

   This method is a factory for the character set instance. The make method takes an integer as input, which corresponds to a character set ID that Oracle supports. For example:

   ```
   int oracleId = CharacterSet.JA16SJIS_CHARSET; // this is character set ID,
                                                 // 832
   ...
   CharacterSet mycharset = CharacterSet.make(oracleId);
   ```

   Each character set that Oracle supports has a unique, predefined Oracle ID.

2. Construct a CHAR object.

   Pass a string, or the bytes that represent the string, to the constructor along with the CharacterSet object that indicates how to interpret the bytes based on the character set. For example:

   ```
   String mystring = "teststring";
   ...
   ```

```
CHAR mychar = new CHAR(teststring, mycharset);
```

There are multiple constructors for `CHAR`, which can take a `String`, a `byte` array, or an object as input along with the `CharacterSet` object. In the case of a `String`, the string is converted to the character set indicated by the `CharacterSet` object before being placed into the `CHAR` object.

Note:

- The `CharacterSet` object cannot be a null value.
- The `CharacterSet` class is an abstract class, therefore it has no constructor. The only way to create instances is to use the `make` method.
- The server recognizes the special value `CharacterSet.DEFAULT_CHARSET` as the database character set. For the client, this value is not meaningful.
- Oracle does not intend or recommend that users extend the `CharacterSet` class.

---

oracle.sql.CHAR Conversion Methods

The `CHAR` class provides the following methods for translating character data to strings:

- `getString`

  This method converts the sequence of characters represented by the `CHAR` object to a string, returning a Java `String` object. If you enter an invalid `OracleID`, then the character set will not be recognized and the `getString` method will throw a `SQLException` exception.

- `toString`

  This method is identical to the `getString` method. But if you enter an invalid `OracleID`, then the character set will not be recognized and the `toString` method will return a hexadecimal representation of the `CHAR` data and will *not* throw a `SQLException` exception.

- `getStringWithReplacement`

  This method is identical to the `getString` method, except a default replacement character replaces characters that have no unicode representation in the `CHAR` object character set. This default character varies from character set to character set, but is often a question mark (`?`).

The database server and the client, or application running on the client, can use different character sets. When you use the methods of the `CHAR` class to transfer data between the server and the client, the JDBC drivers must convert the data from the server character set to the client character set or vice versa. To convert the data, the drivers use Globalization Support.

See Also: Chapter 19, "Globalization Support"

Additional Oracle Type Extensions

Oracle JDBC drivers support the Oracle-specific `BFILE` and `ROWID` data types and `REF CURSOR` types, which are not part of the standard JDBC specification. This section

describes the ROWID and REF CURSOR type extensions. The ROWID is supported as a Java string, and REF CURSOR types are supported as JDBC result sets.

This section covers the following topics:

Oracle ROWID Type

A ROWID is an identification tag unique for each row of an Oracle Database table. The ROWID can be thought of as a virtual column, containing the ID for each row.

The oracle.sql.ROWID class is supplied as a container for ROWID SQL data type.

ROWIDs provide functionality similar to the getCursorName method specified in the java.sql.ResultSet interface and the setCursorName method specified in the java.sql.Statement interface.

If you include the ROWID pseudo-column in a query, then you can retrieve the ROWIDs with the result set getString method. You can also bind a ROWID to a PreparedStatement parameter with the setString method. This enables in-place updating, as in the example that follows.

> **Note:** Use the oracle.sql.ROWID class, only when you are using J2SE 5.0. For JSE 6, you should use the standard java.sql.RowId interface instead.

Example

The following example shows how to access and manipulate ROWID data:

> **Note:** The following example works only with JSE 6.

```
Statement stmt = conn.createStatement();

// Query the employee names with "FOR UPDATE" to lock the rows.
// Select the ROWID to identify the rows to be updated.

ResultSet rset =
   stmt.executeQuery ("SELECT first_name, rowid FROM employees FOR UPDATE");

// Prepare a statement to update the first_name column at a given ROWID

PreparedStatement pstmt =
   conn.prepareStatement ("UPDATE employees SET first_name = ? WHERE rowid = ?");

// Loop through the results of the query
while (rset.next ())
{
    String ename = rset.getString (1);
 RowId rowid = rset.getROWID(2); // Get the ROWID as a String
    pstmt.setString (1, ename.toLowerCase ());
```

```
    pstmt.setROWID (2, rowid); // Pass ROWID to the update statement
    pstmt.executeUpdate ();     // Do the update
}
```

Oracle REF CURSOR Type Category

A cursor variable holds the memory location of a query work area, rather than the contents of the area. Declaring a cursor variable creates a pointer. In SQL, a pointer has the data type `REF x`, where `REF` is short for `REFERENCE` and *x* represents the entity being referenced. A `REF CURSOR`, then, identifies a reference to a cursor variable. Because many cursor variables might exist to point to many work areas, `REF CURSOR` can be thought of as a category or data type specifier that identifies many different types of cursor variables.

> **Note:** `REF CURSOR` instances are not scrollable.

To create a cursor variable, begin by identifying a type that belongs to the `REF CURSOR` category. For example:

```
DECLARE TYPE DeptCursorTyp IS REF CURSOR
```

Then, create the cursor variable by declaring it to be of the type `DeptCursorTyp`:

```
dept_cv DeptCursorTyp  - - declare cursor variable
...
```

`REF CURSOR`, then, is a category of data types, rather than a particular data type.

Stored procedures can return cursor variables of the `REF CURSOR` category. This output is equivalent to a database cursor or a JDBC result set. A `REF CURSOR` essentially encapsulates the results of a query.

In JDBC, a `REF CURSOR` is materialized as a `ResultSet` object and can be accessed as follows:

1. Use a JDBC callable statement to call a stored procedure. It must be a callable statement, as opposed to a prepared statement, because there is an output parameter.
2. The stored procedure returns a `REF CURSOR`.
3. The Java application casts the callable statement to an Oracle callable statement and uses the `getCursor` method of the `OracleCallableStatement` class to materialize the `REF CURSOR` as a JDBC `ResultSet` object.
4. The result set is processed as requested.

> **Important:** The cursor associated with a `REF CURSOR` is closed whenever the statement object that produced the `REF CURSOR` is closed.
>
> Unlike in past releases, the cursor associated with a `REF CURSOR` is *not* closed when the result set object in which the `REF CURSOR` was materialized is closed.

Example

This example shows how to access `REF CURSOR` data.

```
import oracle.jdbc.*;
...
CallableStatement cstmt;
ResultSet cursor;

// Use a PL/SQL block to open the cursor
cstmt = conn.prepareCall
         ("begin open ? for select first_name from employees; end;");

cstmt.registerOutParameter(1, OracleTypes.CURSOR);
cstmt.execute();
cursor = ((OracleCallableStatement)cstmt).getCursor(1);

// Use the cursor like a standard ResultSet
while (cursor.next ())
    {System.out.println (cursor.getString(1));}
```

In the preceding example:

- A `CallableStatement` object is created by using the `prepareCall` method of the connection class.
- The callable statement implements a PL/SQL procedure that returns a `REF CURSOR`.
- As always, the output parameter of the callable statement must be registered to define its type. Use the type code `OracleTypes.CURSOR` for a `REF CURSOR`.
- The callable statement is run, returning the `REF CURSOR`.
- The `CallableStatement` object is cast to `OracleCallableStatement` to use the `getCursor` method, which is an Oracle extension to the standard JDBC API, and returns the `REF CURSOR` into a `ResultSet` object.

Oracle BINARY_FLOAT and BINARY_DOUBLE Types

The Oracle `BINARY_FLOAT` and `BINARY_DOUBLE` types are used to store IEEE 574 float and double data. These correspond to the Java `float` and `double` scalar types with the exception of negative zero and `NaN`.

> **See Also:** *Oracle Database SQL Language Reference*

If you include a `BINARY_DOUBLE` column in a query, then the data is retrieved from the database in the binary format. Also, the `getDouble` method will return the data in the binary format. In contrast, for a `NUMBER` data type column, the number bits are returned and converted to the Java `double` data type.

> **Note:** The Oracle representation for the SQL `FLOAT`, `DOUBLE PRECISION`, and `REAL` data types use the Oracle `NUMBER` representation. The `BINARY_FLOAT` and `BINARY_DOUBLE` data types can be regarded as proprietary types.

A call to the JDBC standard `setDouble(int, double)` method of the `PreparedStatement` interface converts the Java `double` argument to Oracle `NUMBER` style bits and send them to the database. In contrast, the `setBinaryDouble(int, double)` method of the `oracle.jdbc.OraclePreparedStatement` interface converts the data to the internal binary bits and sends them to the database.

You must ensure that the data format used matches the type of the target parameter of the `PreparedStatement` interface. This will result in correct data and least use of CPU. If you use `setBinaryDouble` for a `NUMBER` parameter, then the binary bits are sent to the server and converted to `NUMBER` format. The data will be correct, but server CPU load will be increased. If you use `setDouble` for a `BINARY_DOUBLE` parameter, then the data will first be converted to `NUMBER` bits on the client and sent to the server, where it will be converted back to binary format. This will increase the CPU load on both client and server and can result in data corruption as well.

The `SetFloatAndDoubleUseBinary` connection property when set to `true` causes the JDBC standard APIs, `setFloat(int, float)`, `setDouble(int, double)`, and all the variations, to send internal binary bits instead of `NUBMER` bits.

Note: Although this section largely discusses `BINARY_DOUBLE`, the same is true for `BINARY_FLOAT` as well.

Oracle SYS.ANYTYPE and SYS.ANYDATA Types

Oracle Database 12*c* Release 1 (12.1) provides a Java interface to access the `SYS.ANYTYPE` and `SYS.ANYDATA` Oracle types.

> **See Also:** For information about these Oracle types, refer *Oracle Database PL/SQL Packages and Types Reference*

An instance of the `SYS.ANYTYPE` type contains a type description of any SQL type, persistent or transient, named or unnamed, including object types and collection types. You can use the `oracle.sql.TypeDescriptor` class to access the `SYS.ANYTYPE` type. An `ANYTYPE` instance can be retrieved from a PL/SQL procedure or a SQL `SELECT` statement where `SYS.ANYTYPE` is used as a column type. To retrieve an `ANYTYPE` instance from the database, use the `getObject` method. This method returns an instance of the `TypeDescriptor`.

The retrieved `ANYTYPE` instance could be any of the following:

- Transient object type
- Transient predefined type
- Persistent object type
- Persistent predefined type

Example 4–1 Accessing SYS.ANYTYPE Type

The following code snippet illustrates how to retrieve an instance of `ANYTYPE` from the database:

```
...
ResultSet rs = stmt.executeQuery("select anytype_column from my_table");
TypeDescriptor td = (TypeDescriptor)rs.getObject(1);
short typeCode = td.getInternalTypeCode();
if(typeCode == TypeDescriptor.TYPECODE_OBJECT)
{
  // check if it's a transient type
  if(td.isTransientType())
  {
    AttributeDescriptor[] attributes =
((StructDescriptor)td).getAttributesDescriptor();
    for(int i=0; i<attributes.length; i++)
```

```
      System.out.println(attributes[i].getAttributeName());
  }
  else
  {
    System.out.println(td.getTypeName());
  }
}
...
```

Example 4–2 Creating a Transient Object Type Through PL/SQL and Retrieving Through JDBC

This example provides a code snippet illustrating how to retrieve a transient object type through JDBC.

```
...
OracleCallableStatement cstmt = (OracleCallableStatement)conn.prepareCall
  ("BEGIN ? := transient_obj_type (); END;");
cstmt.registerOutParameter(1,OracleTypes.OPAQUE,"SYS.ANYTYPE");
cstmt.execute();
TypeDescriptor obj = (TypeDescriptor)cstmt.getObject(1);
if(!obj.isTransient())
  System.out.println("This must be a JDBC bug");
cstmt.close();
return obj;
...
```

Example 4–3 Calling a PL/SQL Stored Procedure That Takes an ANYTPE as IN Parameter

The following code snippet illustrates how to call a PL/SQL stored procedure that takes an `ANYTYPE` as `IN` parameter:

```
...
CallableStatement cstmt = conn.prepareCall("BEGIN ? := dumpanytype(?); END;");
cstmt.registerOutParameter(1,OracleTypes.VARCHAR);
// obj is the instance of TypeDescriptor that you have retrieved
cstmt.setObject(2,obj);
cstmt.execute();
String str = (String)cstmt.getObject(1);
...
```

The `oracle.sql.ANYDATA` class enables you to access `SYS.ANYDATA` instances from the database. An instance of this class can be obtained from any valid instance of `oracle.sql.Datum` class. The `convertDatum` factory method takes an instance of `Datum` and returns an instance of `ANYDATA`. The syntax for this factory method is as follows:

```
public static ANYDATA convertDatum(Datum datum) throws SQLException
```

The following is sample code for creating an instance of `oracle.sql.ANYDATA`:

```
// struct is a valid instance of oracle.sql.STRUCT that either comes from the
// database or has been constructed in Java.
ANYDATA myAnyData = ANYDATA.convertDatum(struct);
```

Example 4–4 Accessing an Instance of ANYDATA from the Database

```
...
// anydata_table has been created as:
// CREATE TABLE anydata_tab (data SYS.ANYDATA)
```

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select data from my_anydata_tab");
while(rs.next())
{
  ANYDATA anydata = (ANYDATA)rs.getObject(1);
  if(!anydata.isNull())
  {
    TypeDescriptor td = anydata.getTypeDescriptor();
    if(td.getTypeCode() == OracleType.TYPECODE_OBJECT)
      STRUCT struct = (STRUCT)anydata.accessDatum();
  }
}
...
```

Example 4–5 Inserting an Object as ANYDATA in a Database Table

Consider the following table and object type definition:

```
CREATE TABLE anydata_tab ( id NUMBER, data SYS.ANYDATA)

CREATE OR REPLACE TYPE employee AS OBJECT ( employee_id NUMBER, first_name
VARCHAR2(10) )
```

You can create an instance of the `EMPLOYEE` SQL object type and to insert it into `anydata_tab`le in the following way:

```
...
PreparedStatement pstmt = conn.prepareStatement("insert into anydata_table values
(?,?)");
Struct myEmployeeStr = conn.createStruct("EMPLOYEE", new Object[]{1120,
"Papageno"});
ANYDATA anyda = ANYDATA.convertDatum(myEmployeeStr);
pstmt.setInt(1,123);
pstmt.setObject(2,anyda);
pstmt.executeUpdate();
...
```

Example 4–6 Selecting an ANYDATA Column from a Database Table

```
...
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select data from anydata_table");
while(rs.next())
{
  ANYDATA obj = (ANYDATA)rs.getObject(1);
  TypeDescriptor td = obj.getTypeDescriptor();
}
rs.close();
stmt.close();
...
```

The oracle.jdbc Package

The interfaces of the `oracle.jdbc` package define the Oracle extensions to the interfaces in `java.sql`. These extensions provide access to SQL-format data as described in this chapter. They also provide access to other Oracle-specific functionality, including Oracle performance enhancements.

For the `oracle.jdbc` package, Table 4–1 lists key interfaces and classes used for connections, statements, and result sets.

Table 4–1 Key Interfaces and Classes of the oracle.jdbc Package

Name	Interface or Class	Key Functionality
`OracleDriver`	Class	Implements `java.sql.Driver`
`OracleConnection`	Interface	Provides methods to start and stop an Oracle Database instance and to return Oracle statement objects and methods to set Oracle performance extensions for any statement run in the current connection. Implements `java.sql.Connection`.
`OracleStatement`	Interface	Provides methods to set Oracle performance extensions for individual statement. Is a supertype of `OraclePreparedStatement` and `OracleCallableStatement`. Implements `java.sql.Statement`.
`OraclePreparedStatement`	Interface	Provides `set`*`XXX`* methods to bind `oracle.sql.*` types into a prepared statement. Provides `getMetaData` method to get the metadata from the prepared statements without executing the SELECT statements. Implements `java.sql.PreparedStatement`. Extends `OracleStatement`. Is a supertype of `OracleCallableStatement`.
`OracleCallableStatement`	Interface	Provides `get`*`XXX`* methods to retrieve data in `oracle.sql` format and `set`*`XXX`* methods to bind `oracle.sql.*` types into a callable statement. Implements `java.sql.CallableStatement`. Extends `OraclePreparedStatement`.
`OracleResultSet`	Interface	Provides `get`*`XXX`* methods to retrieve data in `oracle.sql` format. Implements `java.sql.ResultSet`.
`OracleResultSetMetaData`	Interface	Provides methods to get metadata information about Oracle result sets, such as column names and data types. Implements `java.sql.ResultSetMetaData`.

Table 4–1 (Cont.) Key Interfaces and Classes of the oracle.jdbc Package

Name	Interface or Class	Key Functionality
`OracleDatabaseMetaData`	Class	Provides methods to get metadata information about the database, such as database product name and version, table information, and default transaction isolation level. Implements `java.sql.DatabaseMetaData`).
`OracleTypes`	Class	Defines integer constants used to identify SQL types. For standard types, it uses the same values as the standard `java.sql.Types` class. In addition, it adds constants for Oracle extended types.
`OracleArray`	Interface	Includes functionality to retrieve the array as a whole, retrieve a subset of the array elements, and retrieve the SQL base type name of the array elements.
`OracleStruct`	Interface	
`OracleClob`	Interface	
`OracleBlob`	Interface	
`OracleRef`	Interface	
`OracleOpaque`	Interface	

This section covers the following topics:

- Interface oracle.jdbc.OracleConnection
- Interface oracle.jdbc.OracleStatement
- Interface oracle.jdbc.OraclePreparedStatement
- Interface oracle.jdbc.OracleCallableStatement
- Interface oracle.jdbc.OracleResultSet
- Interface oracle.jdbc.OracleResultSetMetaData
- Class oracle.jdbc.OracleTypes

Interface oracle.jdbc.OracleConnection

This interface extends standard JDBC connection functionality to create and return Oracle statement objects, set flags and options for Oracle performance extensions, support type maps for Oracle objects, and support client identifiers.

In Oracle Database 11*g* Release 1, new methods were added to this interface that enable the starting up and shutting down of an Oracle Database instance. Also, for better visibility and clarity, all connection properties are defined as constants in the `OracleConnection` interface.

This interface also defines factory methods for constructing `oracle.sql` data values like `DATE` and `NUMBER`. Remember the following points while using factory methods:

- All code that constructs instances of the `oracle.sql` types should use the Oracle extension factory methods. For example, `ARRAY`, `BFILE`, `DATE`, `INTERVALDS`, `NUMBER`, `STRUCT`, `TIME`, `TIMESTAMP`, and so on.
- All code that constructs instances of the standard types should use the JDBC 4.0 standard factory methods. For example, `CLOB`, `BLOB`, `NCLOB`, and so on.
- There are no factory methods for `CHAR`, `JAVA_STRUCT`, `ArrayDescriptor`, and `StructDescriptor`. These types are for internal driver use only.

> **Note:** Prior to Oracle Database 11*g* Release 1, you had to construct `ArrayDescriptors` and `StructDescriptors` for passing as arguments to the `ARRAY` and `STRUCT` class constructors. The new `ARRAY` and `Struct` factory methods do not have any descriptor arguments. The driver still uses descriptors internally, but you do not need to create them.

Client Identifiers

In a connection pooling environment, the client identifier can be used to identify the lightweight user using the database session currently. A client identifier can also be used to share the Globally Accessed Application Context between different database sessions. The client identifier set in a database session is audited when database auditing is turned on.

> **See Also:** *Oracle Database Development Guide* and *Oracle Database JDBC Java API Reference*

Interface oracle.jdbc.OracleStatement

This interface extends standard JDBC statement functionality and is the superinterface of the `OraclePreparedStatement` and `OracleCallableStatement` classes. Extended functionality includes support for setting flags and options for Oracle performance extensions on a statement-by-statement basis, as opposed to the `OracleConnection` interface that sets these on a connectionwide basis.

Interface oracle.jdbc.OraclePreparedStatement

This interface extends the `OracleStatement` interface and extends standard JDBC prepared statement functionality. Also, the `oracle.jdbc.OraclePreparedStatement` interface is extended by the `OracleCallableStatement` interface. Extended functionality consists of the following:

- `set`*`XXX`* methods for binding `oracle.sql.*` types and objects to prepared statements
- `getMetaData` method to get the metadata from the prepared statements without executing the SELECT statements
- Methods to support Oracle performance extensions on a statement-by-statement basis

> **Note:** Do not use the `PreparedStatement` interface to create a trigger that refers to a `:NEW` or `:OLD` column. Use `Statement` instead. Using `PreparedStatement` will cause execution to fail with the message `java.sql.SQLException: Missing IN or OUT parameter at index:: 1`.

Interface oracle.jdbc.OracleCallableStatement

This interface extends the `OraclePreparedStatement` interface, which extends the `OracleStatement` interface and incorporates standard JDBC callable statement functionality.

Note: Do not use the `CallableStatement` interface to create a trigger that refers to a `:NEW` or `:OLD` column. Use `Statement` instead; using `CallableStatement` will cause execution to fail with the message `java.sql.SQLException: Missing IN or OUT parameter at index::1`

Note:

- The `setXXX(String,...)` and `registerOutParameter(String,...)` methods can be used only if all binds are procedure or function parameters only. The statement can contain no other binds and the parameter binds must be indicated with a question mark (`?`) and not `:XX`.
- If you are using `setXXX(int,...)` or `setXXXAtName(String,...)` method, then any output parameter is bound with `registerOutParameter(int,...)` and not `registerOutParameter(String,...)`, which is for named parameter notation.

Interface oracle.jdbc.OracleResultSet

This interface extends standard JDBC result set functionality, implementing `getXXX` methods for retrieving data into `oracle.sql.*` objects.

Interface oracle.jdbc.OracleResultSetMetaData

This interface extends standard JDBC result set metadata functionality to retrieve information about Oracle result set objects.

See Also: "Using Result Set Metadata Extensions" on page 11-14

Class oracle.jdbc.OracleTypes

The `OracleTypes` class defines constants that JDBC uses to identify SQL types. Each variable in this class has a constant integer value. The `oracle.jdbc.OracleTypes` class duplicates the type code definitions of the standard Java `java.sql.Types` class and contains these additional type codes for Oracle extensions:

- `OracleTypes.BFILE`
- `OracleTypes.ROWID`
- `OracleTypes.CURSOR` (for `REF CURSOR` types)
- `OracleTypes.CHAR_BYTES` (for calling `setNull` and `setCHAR` methods on the same column)

As in `java.sql.Types`, all the variable names are in uppercase text.

JDBC uses the SQL types identified by the elements of the `OracleTypes` class in two main areas: registering output parameters and in the `setNull` method of the `PreparedStatement` class.

OracleTypes and Registering Output Parameters

The type codes in `java.sql.Types` or `oracle.jdbc.OracleTypes` identify the SQL types of the output parameters in the `registerOutParameter` method of the `java.sql.CallableStatement` and `oracle.jdbc.OracleCallableStatement` interfaces.

These are the forms that the `registerOutputParameter` method can take for the `CallableStatement` and `OracleCallableStatement` interfaces

```
cs.registerOutParameter(int index, int sqlType);

cs.registerOutParameter(int index, int sqlType, String sql_name);

cs.registerOutParameter(int index, int sqlType, int scale);
```

In these signatures, `index` represents the parameter index, `sqlType` is the type code for the SQL data type, `sql_name` is the name given to the data type, for user-defined types, when `sqlType` is a `STRUCT`, `REF`, or `ARRAY` type code, and `scale` represents the number of digits to the right of the decimal point, when `sqlType` is a `NUMERIC` or `DECIMAL` type code.

The following example uses a `CallableStatement` interface to call a procedure named `charout`, which returns a `CHAR` data type. Note the use of the `OracleTypes.CHAR` type code in the `registerOutParameter` method.

```
CallableStatement cs = conn.prepareCall ("BEGIN charout (?); END;");
cs.registerOutParameter (1, OracleTypes.CHAR);
cs.execute ();
System.out.println ("Out argument is: " + cs.getString (1));
```

The next example uses a `CallableStatement` interface to call `structout`, which returns a `STRUCT` data type. The form of `registerOutParameter` requires you to specify the type code, `Types.STRUCT` or `OracleTypes.STRUCT`, as well as the SQL name, `EMPLOYEE`.

The example assumes that no type mapping has been declared for the `EMPLOYEE` type, so it is retrieved into a `STRUCT` data type. To retrieve the value of `EMPLOYEE` as an `oracle.sql.STRUCT` object, the statement object `cs` is cast to `OracleCallableStatement` and the Oracle extension `getSTRUCT` method is invoked.

```
CallableStatement cs = conn.prepareCall ("BEGIN structout (?); END;");
cs.registerOutParameter (1, OracleTypes.STRUCT, "EMPLOYEE");
cs.execute ();

// get the value into a STRUCT because it
// is assumed that no type map has been defined
STRUCT emp = ((OracleCallableStatement)cs).getSTRUCT (1);
```

OracleTypes and the setNull Method

The type codes in `Types` and `OracleTypes` identify the SQL type of the data item, which the `setNull` method sets to `NULL`. The `setNull` method can be found in the `java.sql.PreparedStatement` and `oracle.jdbc.OraclePreparedStatement` interfaces.

These are the forms that the `setNull` method can take for the `PreparedStatement` and `OraclePreparedStatement` objects:

```
ps.setNull(int index, int sqlType);

ps.setNull(int index, int sqlType, String sql_name);
```

In these signatures, `index` represents the parameter index, `sqlType` is the type code for the SQL data type, and `sql_name` is the name given to the data type, for user-defined

types, when sqlType is a STRUCT, REF, or ARRAY type code. If you enter an invalid sqlType, a ParameterTypeConflict exception is thrown.

The following example uses a prepared statement to insert a null value into the database. Note the use of OracleTypes.NUMERIC to identify the numeric object set to NULL. Alternatively, Types.NUMERIC can be used.

```
PreparedStatement pstmt =
    conn.prepareStatement ("INSERT INTO num_table VALUES (?)");

pstmt.setNull (1, OracleTypes.NUMERIC);
pstmt.execute ();
```

In this example, the prepared statement inserts a NULL STRUCT object of type EMPLOYEE into the database.

```
PreparedStatement pstmt = conn.prepareStatement
                               ("INSERT INTO employees VALUES (?)");

pstmt.setNull (1, OracleTypes.STRUCT, "EMPLOYEE");
pstmt.execute ();
```

You can also use the OracleTypes.CHAR_BYTES type with the setNull method, if you also want to call the setCHAR method on the same column. For example:

```
 ps.setCHAR(n, aCHAR);
  ps.addBatch();
  ps.setNull(n, OracleTypes.CHAR_BYTES);
  ps.addBatch();
```

In this preceding example, any other type, apart from the OracleTypes.CHAR_BYTES type, will cause extra round trips to the Database. Alternatively, you can also write your code without using the setNull method. For example, you can also write your code as shown in the following example:

```
ps.setCHAR(n, null);
```

DML Returning

The DML returning feature provides more functionality compared to retrieval of auto-generated keys. It can be used to retrieve not only auto-generated keys, but also other columns or values that the application may use.

Note:

- The server-side internal driver does not support DML returning and retrieval of auto-generated keys.
- You cannot use both DML returning and retrieval of auto-generated keys in the same statement.

The following sections explain the support for DML returning:

- Oracle-Specific APIs
- Running DML Returning Statements
- Example of DML Returning
- Limitations of DML Returning

See Also: "Retrieval of Auto-Generated Keys" on page 3-4

Oracle-Specific APIs

The `OraclePreparedStatement` interface is enhanced with Oracle-specific application programming interfaces (APIs) to support DML returning. The `registerReturnParameter` and `getReturnResultSet` methods have been added to the `oracle.jdbc.OraclePreparedStatement` interface, to register parameters that are returned and data retrieved by DML returning.

The `registerReturnParameter` method is used to register the return parameter for DML returning. The method throws a `SQLException` instance if an error occurs. You must pass a positive integer specifying the index of the return parameter. You also must specify the type of the return parameter. You can also specify the maximum bytes or characters of the return parameter. This method can be used only with `char` or `RAW` types. You can also specify the fully qualified name of a SQL structure type.

Note: If you do not know the maximum size of the return parameters, then you should use `registerReturnParameter(int paramIndex, int externalType)`, which picks the default maximum size. If you know the maximum size of return parameters, using `registerReturnParameter(int paramIndex, int externalType, int maxSize)` can reduce memory consumption.

The `getReturnResultSet` method fetches the data returned from DML returning and returns it as a `ResultSet` object. The method throws a `SQLException` exception if an error occurs.

Note: The Oracle-specific APIs for the DML returning feature are in `ojdbc6.jar` for Java Development Kit (JDK) 6.0 and in `ojdbc7.jar` for JDK 7.

Running DML Returning Statements

Before running a DML returning statement, the JDBC application must call one or more of the `registerReturnParameter` methods. The method provides the JDBC drivers with information, such as type and size, of the return parameters. The DML returning statement is then processed using one of the standard JDBC APIs, `executeUpdate` or `execute`. You can then fetch the returned parameters as a `ResultSet` object using the `getReturnResultSet` method of the `oracle.jdbc.OraclePreparedStatement` interface.

In order to read the values in the `ResultSet` object, the underlying `Statement` object must be open. When the underlying `Statement` object is closed, the returned `ResultSet` object is also closed. This is consistent with `ResultSet` objects that are retrieved by processing SQL query statements.

When a DML returning statement is run, the concurrency of the `ResultSet` object returned by the `getReturnResultSet` method must be `CONCUR_READ_ONLY` and the type of the `ResultSet` object must be `TYPE_FORWARD_ONLY` or `TYPE_SCROLL_INSENSITIVE`.

Example of DML Returning

This section provides two code examples of DML returning.

The following code example illustrates the use of DML returning. In this example, assume that the maximum size of the `name` column is 100 characters. Because the maximum size of the `name` column is known, the `registerReturnParameter(int paramIndex, int externalType, int maxSize)` method is used.

```
...
OraclePreparedStatement pstmt = (OraclePreparedStatement)conn.prepareStatement(
       "delete from tab1 where age < ? returning name into ?");
pstmt.setInt(1,18);

/** register returned parameter
  * in this case the maximum size of name is 100 chars
  */
pstmt.registerReturnParameter(2, OracleTypes.VARCHAR, 100);

// process the DML returning statement
count = pstmt.executeUpdate();
if (count>0)
{
  ResultSet rset = pstmt.getReturnResultSet(); //rest is not null and not empty
  while(rset.next())
  {
    String name = rset.getString(1);
    ...
  }
}
...
```

The following code example also illustrates the use of DML returning. However, in this case, the maximum size of the return parameters is not known. Therefore, the `registerReturnParameter(int paramIndex, int externalType)` method is used.

```
...
OraclePreparedStatement pstmt = (OraclePreparedStatement)conn.prepareStatement(
  "insert into lobtab values (100, empty_clob()) returning col1, col2 into ?, ?");

// register return parameters
pstmt.registerReturnParameter(1, OracleTypes.INTEGER);
pstmt.registerReturnParameter(2, OracleTypes.CLOB);

// process the DML returning SQL statement
pstmt.executeUpdate();
ResultSet rset = pstmt.getReturnResultSet();
int r;
CLOB clob;
if (rset.next())
{
  r = rset.getInt(1);
  System.out.println(r);
  clob = (CLOB)rset.getClob(2);
  ...
}
...
```

Limitations of DML Returning

When using DML returning, be aware of the following:

- It is unspecified what the `getReturnResultSet` method returns when it is invoked more than once. You should not rely on any specific action in this regard.
- The `ResultSet` objects returned from the execution of DML returning statements do not support the `ResultSetMetaData` type. Therefore, the applications must know the information of return parameters before running DML returning statements.
- Streams are not supported with DML returning.
- DML returning cannot be combined with batch update.
- You cannot use both the auto-generated key feature and the DML returning feature in a single SQL DML statement. For example, the following is not allowed:

```
...
PreparedStatement pstmt = conn.prepareStatement('insert into orders (?, ?, ?)
returning order_id into ?");
pstmt.setInt(1, seq01.NEXTVAL);
pstmt.setInt(2, 100);
pstmt.setInt(3, 966431502);
pstmt.registerReturnParam(4, OracleTypes.INTEGER);
pstmt.executeUpdate;
ResultSet rset = pstmt.getGeneratedKeys;
...
```

Accessing PL/SQL Associative Arrays

Oracle JDBC drivers enable JDBC applications to make PL/SQL calls with Associative Arrays parameters. In PL/SQL, an Associative Array is a set of key-value pairs, where the keys may be `PLS_INTEGERs` or Strings. The keys may have any value and need not be dense. From a client application, you can work only with `PLS_INTEGER` keys that must be positive and dense.

> **Note:** Associative Arrays were previously known as index-by tables.

This section covers the following topics:

Overview

Oracle JDBC drivers support PL/SQL Associative Arrays of `VARCHAR` and `NUMBER` types. Typical Oracle JDBC input binding, output registration, and data access methods do not support PL/SQL Associative Arrays. This section discusses the additional methods to support these types.

The `OraclePreparedStatement` and `OracleCallableStatement` classes define the additional methods. These methods include the following:

- `setPlsqlIndexTable`
- `registerIndexTableOutParameter`

- `getOraclePlsqlIndexTable`
- `getPlsqlIndexTable`

These methods handle PL/SQL Associative Arrays as `IN`, `OUT`, or `IN OUT` parameters, including function return values.

Note: When you use String data types, the size is limited to the size in PL/SQL that is 32767 characters. For the server-side internal driver, the limits are lower. Refer to the Javadoc for more information about these methods.

See Also: *Oracle Database PL/SQL Language Reference*

Binding IN Parameters

To bind a PL/SQL Associative Array parameter in the `IN` parameter mode, use the `setPlsqlIndexTable` method defined in the `OraclePreparedStatement` and `OracleCallableStatement` classes.

```
synchronized public void setPlsqlIndexTable (int paramIndex, Object arrayData, int
maxLen, int curLen, int elemSqlType,
 int elemMaxLen) throws SQLException
```

Table 4–2 describes the arguments of the `setPlsqlIndexTable` method.

Table 4–2 Arguments of the setPlsqlIndexTable Method

Argument	Description
`int paramIndex`	Indicates the parameter position within the statement.
`Object arrayData`	Is an array of values to be bound to the PL/SQL Associative Array parameter. The value is of type `java.lang.Object`. The value can be a Java primitive type array, such as `int[]`, or a Java object array, such as `BigDecimal[]`.
`int maxLen`	Specifies the maximum table length of the Associative Array bind value that defines the maximum possible `curLen` for batch updates. For standalone binds, `maxLen` should use the same value as `curLen`. This argument is required.
`int curLen`	Specifies the actual size of the Associative Array bind value in `arrayData`. If the `curLen` value is smaller than the size of `arrayData`, then only the `curLen` number of table elements is passed to the database. If the `curLen` value is larger than the size of `arrayData`, then the entire `arrayData` is sent to the database.
`int elemSqlType`	Specifies the Associative Array element type based on the values defined in the `OracleTypes` class.
`int elemMaxLen`	Specifies the Associative Array element maximum length in case the element type is `CHAR`, `VARCHAR`, or `RAW`. This value is ignored for other types.

The following code example uses the `setPlsqlIndexTable` method to bind an Associative Array as an `IN` parameter:

```
// Prepare the statement
OracleCallableStatement procin = (OracleCallableStatement)
   conn.prepareCall ("begin procin (?); end;");

// Associative Array bind value
```

```
int[] values = { 1, 2, 3 };

// maximum length of the Associative Array bind value. This
// value defines the maximum possible "currentLen" for batch
// updates. For standalone binds, "maxLen" should be the
// same as "currentLen".
int maxLen = values.length;

// actual size of the Associative Array bind value
int currentLen = values.length;

// Associative Array element type
int elemSqlType = OracleTypes.NUMBER;

// Associative Array element length in case the element type
// is CHAR, VARCHAR or RAW. This value is ignored for other
// types.
int elemMaxLen = 0;

// set the value
procin.setPlsqlIndexTable (1, values,
                           maxLen, currentLen,
                           elemSqlType, elemMaxLen);

// execute the call
procin.execute ();
```

Receiving OUT Parameters

This section describes how to register a PL/SQL Associative Array as an `OUT` parameter. In addition, it describes how to access the `OUT` bind values in various mapping styles.

> **Note:** The methods described in this section apply to function return values and the `IN OUT` parameter mode as well.

Registering the OUT Parameters

To register a PL/SQL Associative Array as an `OUT` parameter, use the `registerIndexTableOutParameter` method defined in the `OracleCallableStatement` class.

```
synchronized public void registerIndexTableOutParameter
   (int paramIndex, int maxLen, int elemSqlType,
 int elemMaxLen) throws SQLException
```

Table 4–3 describes the arguments of the `registerIndexTableOutParameter` method.

Table 4–3 Arguments of the registerIndexTableOutParameter Method

Argument	Description
`int paramIndex`	Indicates the parameter position within the statement.
`int maxLen`	Specifies the maximum table length of the Associative Array bind value to be returned.
`int elemSqlType`	Specifies the Associative Array element type based on the values defined in the `OracleTypes` class.

Table 4–3 (Cont.) Arguments of the registerIndexTableOutParameter Method

Argument	Description
`int elemMaxLen`	Specifies the Associative Array element maximum length in case the element type is `CHAR`, `VARCHAR`, or `FIXED_CHAR`. This value is ignored for other types.

The following code example uses the `registerIndexTableOutParameter` method to register an Associative Array as an `OUT` parameter:

```
// maximum length of the Associative Array value. This
// value defines the maximum table size to be returned.
int maxLen = 10;

// Associative Array element type
int elemSqlType = OracleTypes.NUMBER;

// Associative Array element length in case the element type
// is CHAR, VARCHAR or FIXED_CHAR. This value is ignored for other
// types
int elemMaxLen = 0;

// register the return value
funcnone.registerIndexTableOutParameter
   (1, maxLen, elemSqlType, elemMaxLen);
```

Accessing the OUT Parameter Values

To access the `OUT` bind value, the `OracleCallableStatement` class defines multiple methods that return the Associative Array values in different mapping styles. There are three mapping choices available in JDBC drivers:

Mappings	Methods to Use
JDBC default mappings	`getPlsqlIndexTable(int)`
Oracle mappings	`getOraclePlsqlIndexTable(int)`
Java primitive type mappings	`getPlsqlIndexTable(int, Class)`

Type Mappings

This section covers the following topics:

- JDBC Default Mappings
- Oracle Mappings
- Java Primitive Type Mappings

JDBC Default Mappings

The `getPlsqlIndexTable(int)` method returns Associative Array elements using the JDBC default mappings. The syntax for this method is the following:

```
public Object getPlsqlIndexTable (int paramIndex)
   throws SQLException
```

Table 4–4 describes the argument of the `getPlsqlIndexTable` method.

Table 4–4 Argument of the getPlsqlIndexTable Method

Argument	Description
`int paramIndex`	This argument indicates the parameter position within the statement.

The return value is a Java array. The elements of this array are of the default Java type corresponding to the SQL type of the elements. For example, for an Associative Array with elements of `NUMERIC` type code, the element values are mapped to `BigDecimal` by Oracle JDBC driver, and the `getPlsqlIndexTable` method returns a `BigDecimal[]` array. For a JDBC application, you must cast the return value to `BigDecimal[]` to access the table element values.

The following code example uses the `getPlsqlIndexTable` method to return Associative Array elements with JDBC default mapping:

```
// access the value using JDBC default mapping
BigDecimal[] values =
   (BigDecimal[]) procout.getPlsqlIndexTable (1);

// print the elements
for (int i=0; i<values.length; i++)
   System.out.println (values[i].intValue());
```

Oracle Mappings

The `getOraclePlsqlIndexTable` method returns Associative Array elements using Oracle mapping.

```
public Datum[] getOraclePlsqlIndexTable (int paramIndex)
      throws SQLException
```

Table 4–5 describes the argument of the `getOraclePlsqlIndexTable` method.

Table 4–5 Argument of the getOraclePlsqlIndexTable Method

Argument	Description
`int paramIndex`	Indicates the parameter position within the statement.

The return value is an `oracle.sql.Datum` array, and the elements in the array are of the default `Datum` type corresponding to the SQL type of the element. For example, the element values of an Associative Array of numeric elements are mapped to the `oracle.sql.NUMBER` type in Oracle mapping, and the `getOraclePlsqlIndexTable` method returns an `oracle.sql.Datum` array that contains `oracle.sql.NUMBER` elements.

The following code example uses the `getOraclePlsqlIndexTable` method to access the elements of a PL/SQL Associative Array `OUT` parameter, using Oracle mapping:

```
// Prepare the statement
OracleCallableStatement procout = (OracleCallableStatement)
                                  conn.prepareCall ("begin procout (?); end;");

...

// run the call
procout.execute ();

// access the value using Oracle JDBC mapping
```

```
Datum[] outvalues = procout.getOraclePlsqlIndexTable (1);

// print the elements
for (int i=0; i<outvalues.length; i++)
   System.out.println (outvalues[i].intValue());
```

Java Primitive Type Mappings

The `getPlsqlIndexTable(int, Class)` method returns Associative Array elements in Java primitive types. The return value is a Java array. The syntax for this method is the following:

```
synchronized public Object getPlsqlIndexTable
   (int paramIndex, Class primitiveType) throws SQLException
```

Table 4–6 describes the arguments of the `getPlsqlIndexTable` method.

Table 4–6 Arguments of the getPlsqlIndexTable Method

Argument	Description
`int paramIndex`	Indicates the parameter position within the statement.

`Class primitiveType`	Specifies a Java primitive type to which the Associative Array elements are to be converted. For example, if you specify `java.lang.Integer.TYPE`, the return value is an `int` array.
	The following are the possible values of this parameter:
	`java.lang.Integer.TYPE`
	`java.lang.Long.TYPE`
	`java.lang.Float.TYPE`
	`java.lang.Double.TYPE`
	`java.lang.Short.TYPE`

The following code example uses the `getPlsqlIndexTable` method to access the elements of a PL/SQL Associative Array of numbers. In the example, the second parameter specifies `java.lang.Integer.TYPE` and the return value of the `getPlsqlIndexTable` method is an `int` array.

```
OracleCallableStatement funcnone = (OracleCallableStatement)
   conn.prepareCall ("begin ? := funcnone; end;");

// maximum length of the Associative Array value. This
// value defines the maximum table size to be returned.
int maxLen = 10;

// Associative Array element type
int elemSqlType = OracleTypes.NUMBER;

// Associative Array element length in case the element type
// is CHAR, VARCHAR or RAW. This value is ignored for other
// types
int elemMaxLen = 0;

// register the return value
funcnone.registerIndexTableOutParameter (1, maxLen,
                                         elemSqlType, elemMaxLen);
// execute the call
funcnone.execute ();
```

```
// access the value as a Java primitive array.
int[] values = (int[])
   funcnone.getPlsqlIndexTable (1, java.lang.Integer.TYPE);

// print the elements
for (int i=0; i<values.length; i++)
   System.out.println (values[i]);
```

5

Features Specific to JDBC Thin

This chapter introduces the Java Database Connectivity (JDBC) Thin client and covers the features supported only by the JDBC Thin driver. It also provides basic information about working with Oracle JDBC applets. This following topics are covered in this chapter:

- Overview of JDBC Thin Client
- Additional Features Supported
- JDBC in Applets

Overview of JDBC Thin Client

The JDBC Thin client is a pure Java, Type IV driver. It is lightweight and easy to install. It provides high performance, comparable to the performance provided by the JDBC Oracle Call Interface (OCI) driver. The JDBC Thin driver is written entirely in Java, and therefore, it is platform-independent. Also, this driver does not require any additional Oracle software on the client-side.

The JDBC Thin driver communicates with the server using TTC, a protocol developed by Oracle to access data from Oracle Database. It can be used for application servers as well as for applets. The driver allows a direct connection to the database by providing an implementation of TCP/IP that implements Oracle Net and TTC on top of Java sockets. Both of these protocols are lightweight implementation versions of their counterparts on the server. The Oracle Net protocol runs over TCP/IP only.

The JDBC Thin driver can be used on both the client-side and the server-side. On the client-side, drivers can be used in Java applications or Java applets that run either on the client or in the middle tier of a three-tier configuration. On the server-side, this driver is used to access a remote Oracle Database instance or another session on the same database.

Additional Features Supported

The JDBC Thin driver supports all standard JDBC features. The JDBC Thin driver also provides support for the following additional features:

- Default Support for Native XA
- Support for Transaction Guard
- Support for Application Continuity
- Support for Applets

Default Support for Native XA

Similar to the JDBC OCI driver, the JDBC Thin driver also provides support for Native XA. However, the JDBC Thin driver provides support for Native XA by default. This is unlike the case of the JDBC OCI driver, in which the support for Native XA is not enabled by default.

See Also: "Native-XA in Oracle JDBC Drivers" on page 30-19

Support for Transaction Guard

Transaction Guard feature provides a generic infrastructure for at-most-once execution during planned and unplanned outages and duplicate submissions. Transaction Guard feature (along with Application Continuity feature) provides transparent session recovery and replay of SQL statements (queries and DMLs) since the beginning of the in-flight transaction.

See Also: Chapter 26, "Transaction Guard for Java"

Support for Application Continuity

Application Continuity provides a general purpose, application-independent infrastructure that enables recovery of work from an application perspective, after the occurrence of a planned or unplanned outage. It provides the following benefits:

- Masking of outages from the end user
- Recovery of user environments, in-flight transactions, and lost outcome
- A single, easy, and foolproof method for applications to recover
- A definite target response time for applications, regardless of outages

See Also: Chapter 27, "Application Continuity for Java"

Support for Applets

The JDBC Thin driver is the only Oracle JDBC driver that provides support for applets. This driver can be downloaded along with the Java applet that is being run in a browser.

Note: When the JDBC Thin driver is used with an applet, the browser used on the client-side must have the capability to support Java sockets.

The HTTP protocol, which is usually used for communication over a network, is stateless. However, the JDBC Thin driver is not stateless. Therefore, the initial HTTP request to download the applet and the JDBC Thin driver is stateless. After the JDBC Thin driver establishes the database connection, the communication between the browser and the database is stateful and in a two-tier configuration.

See Also: "JDBC in Applets" on page 5-2

JDBC in Applets

You can use only the Oracle JDBC Thin driver for an applet. This section describes what you must do to connect an applet to a database. This description includes how to use the Connection Manager feature of Oracle Database, or signed applets if you are

connecting to a database that is running on a different host from the Web server. It also describes how your applet can connect to a database through a firewall. The section concludes with how to package and deploy the applet.

The following topics are covered:

- Connecting to the Database Through the Applet
- Connecting to a Database on a Different Host Than the Web Server
- Using Applets with Firewalls
- Packaging Applets
- Specifying an Applet in an HTML Page

Connecting to the Database Through the Applet

The most common task of an applet using the JDBC driver is to connect to and query a database. Because of applet security restrictions, unless particular steps are taken, an applet can open TCP/IP sockets only to the host from which it was downloaded. This is the host on which the Web server is running. This means that without these steps, your applet can connect only to a database that is running on the same host as the Web server.

If your database and Web server are running on the same host, then there is no issue and no special steps are required. You can connect to the database as you would from an application.

As with connecting from an application, there are two ways in which you can specify the connection information to the driver. You can provide it in the form of `host:port:service_name` or in the form of TNS keyword-value syntax.

For example, if the database to which you want to connect resides on the `localost`, at port `5221`, and service name `orcl`, and you want to connect with user name `HR` and password `hr`, then use either of the two following connection strings:

- Using `host:port:service_name` syntax:

```
String connString="jdbc:oracle:thin:@localhost:5221:orcl";

OracleDataSource ods = new OracleDataSource();
ods.setURL(connString);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

- Using TNS keyword-value syntax:

```
String connString = "jdbc:oracle:thin:@(description=(address_
list=(address=(protocol=tcp)
(port=5221)(host=localHost)))(connect_data=(INSTANCE_NAME=orcl)))";
OracleDataSource ods = new OracleDataSource();

ods.setURL(connString);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

If you use the TNS keyword-value pair to specify the connection information to the JDBC Thin driver, then you must declare the protocol as TCP.

However, a Web server and database server both require many resources. You seldom find both servers running on the same computer. Usually, your applet connects to a database on a host other than the one on which the Web server runs. If you want your applet to connect to a database running on a different computer, then you have the following options:

- Use the Oracle Connection Manager on the host computer. The applet can connect to the Connection Manager, which connects to a database on another computer.
- Use signed applets, which can request socket connection privileges to other computers.

Your applet can also take advantage of the data encryption and integrity checksum features of the Advanced Security option of Oracle Database.

Connecting to a Database on a Different Host Than the Web Server

If you are connecting to a database on a host other than the one on which the Web server is running, then you must overcome applet security restrictions. You can do this in the following ways:

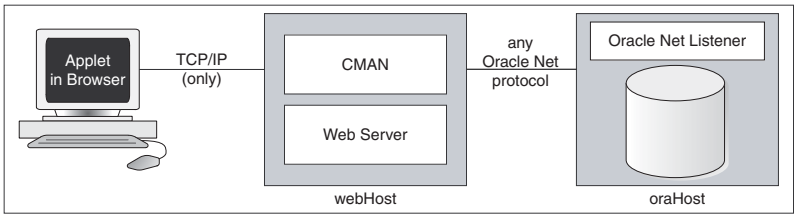
- Using the Oracle Connection Manager
- Using Signed Applets

Using the Oracle Connection Manager

The Oracle Connection Manager is a lightweight, highly scalable program that can receive Oracle Net packets and retransmit them to a different server. To a client running Oracle Net, the Connection Manager looks exactly like a database server. An applet that uses the JDBC Thin driver can connect to a Connection Manager running on the Web server host and have the Connection Manager redirect the Oracle Net packets to an Oracle server running on a different host.

Figure 5–1 illustrates the relationship between the applet, the Oracle Connection Manager, and the database.

Figure 5–1 Applet, Connection Manager, and Database Relationship



Using the Oracle Connection Manager requires two steps:

- Install and run the Connection Manager.
- Write the connection string that targets the Connection Manager.

Installing and Running the Oracle Connection Manager

You must install the Connection Manager, available on the Oracle distribution media, onto the Web server host.

On the Web server host, create a `CMAN.ORA` file in the *`ORACLE_HOME`*`/NET8/ADMIN` directory. The options you can declare in a `CMAN.ORA` file include firewall and connection pooling support.

Here is an example of a very simple CMAN.ORA file. Replace *web-server-host* with the name of your Web server host. The fourth line in the file indicates that the Connection Manager is listening on port 1610. You must use the same port number in your connection string for JDBC.

```
cman = (ADDRESS_LIST =
       (ADDRESS = (PROTOCOL=TCP)
       (HOST=web-server-host)
       (PORT=1610)))

cman_profile = (parameter_list =
       (MAXIMUM_RELAYS=512)
       (LOG_LEVEL=1)
   (TRACING=YES)
       (RELAY_STATISTICS=YES)
       (SHOW_TNS_INFO=YES)
       (USE_ASYNC_CALL=YES)
       (AUTHENTICATION_LEVEL=0)
       )
```

After you create the file, start the Connection Manager at the operating system prompt with the following command:

```
cmctl start
```

> **Note:** While installing Oracle Connection Manager, if you choose to run Oracle Connection Manager services as an authenticated user, then the cmctl command asks for a password. But, if you choose to run Oracle Connection Manager services as a local service account, then the cmctl command does not ask for a password.

To use your applet, you must now write the connection string for it.

Writing the URL that Targets the Connection Manager

The following text describes how to write the URL in your applet, so that the applet connects to the Connection Manager and the Connection Manager connects with the database. In the URL, you specify an address list that lists the protocol, port, and name of the Web server host on which the Connection Manager is running, followed by the protocol, port, and name of the host on which the database is running.

The following example describes the configuration illustrated in Figure 5–1. The Web server on which the Connection Manager is running is on host webHost and is listening on port 1610. The database to which you want to connect is running on host oraHost, listening on port 5221, and service name orcl. You write the URL in TNS keyword-value format:

```
String myURL =
   "jdbc:oracle:thin:@(description=(address_list=
   (address=(protocol=tcp)(port=1610)(host=webHost))
   (address=(protocol=tcp)(port=5221)(host=oraHost)))
   (connect_data=(INSTANCE_NAME=orcl))
   (source_route=yes))";
  OracleDataSource ods = new OracleDataSource();
  ods.setURL(myURL);
  ods.setUser("HR");
  ods.setPassword("hr");
  Connection conn = ods.getConnection();
```

The first element in the `address_list` entry represents the connection to the Connection Manager. The second element represents the database to which you want to connect. The order in which you list the addresses is important.

When your applet uses a URL, such as the preceding one, it will function exactly as if it were connected directly to the database on the host `oraHost`.

Connecting Through Multiple Connection Managers

Your applet can reach its target database even if it first has to go through multiple Connection Managers. For example, if the Connection Managers form a proxy chain. To do this, add the addresses of the Connection Managers to the address list, in the order that you plan to access them. The database listener should be the last address on this list.

Using Signed Applets

In a Java Development Kit (JDK) 1.2.x-based or later browser, an applet can request socket connection privileges and connect to a database running on a different host than the Web server host. Starting from Netscape 4.0, you perform this by signing your applet, that is, writing a signed applet. You must follow these steps:

1. Sign the applet. For information about the steps you must follow to sign an applet, refer to

   http://www.oracle.com/technetwork/java/index.htm

2. Include applet code that asks for appropriate privileges before opening a socket.

   If you are using Netscape, then your code would include a statement like this:

   ```
   netscape.security.PrivilegeManager.enablePrivilege("UniversalConnect");
   OracleDataSource ods = new OracleDataSource();
   ods.setURL("jdbc:oracle:thin:HR/hr@localhost:5221:orcl");
   Connection conn = ods.getConnection();
   ```

3. You must obtain an object-signing certificate. Refer to a site that provides information about obtaining and installing a certificate.

For information about the Java Security API, including signed applet examples, see the following site:

http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

Using Applets with Firewalls

Under standard circumstances, an applet that uses the JDBC Thin driver cannot access the database through a firewall. In general, the purpose of a firewall is to prevent unauthorized clients from reaching the server. In the case of applets trying to connect to the database, the firewall prevents the opening of a TCP/IP socket to the database.

In general, firewalls are rule-based. They have a list of rules that define which clients can connect, and which cannot. Firewalls compare the host name of the client with the rules and, based on this comparison, either grant the client access or deny access. If the host name lookup fails, then the firewall tries again. This time, the firewall extracts the IP address of the client and compares it to the rules. The firewall is designed to do this so that users can specify rules that include host names as well as IP addresses.

You can solve the firewall issue by using an Oracle Net-compliant firewall and connection strings that comply with the firewall configuration. Oracle Net-compliant firewalls are available from many leading vendors.

An unsigned applet can access only the same host from which it is downloaded. In this case, the Oracle Net-compliant firewall must be installed on that host. In contrast, a signed applet can connect to any host. In this case, the firewall on the target host controls the access.

Connecting through a firewall requires two steps, as described in the following sections:

- Configuring a Firewall for Applets that use the JDBC Thin Driver
- Writing a URL to Connect Through a Firewall

Configuring a Firewall for Applets that use the JDBC Thin Driver

The instructions in this section assume that you are running an Oracle Net-compliant firewall.

Java applets do not have access to the local system. Because of the security limitations, applets cannot access the host name or environment variables on the local system. As a result, the JDBC Thin driver cannot access the host name on which it is running. The firewall cannot be provided with the host name. To allow requests from JDBC Thin clients to go through the firewall, you must do the following to the list of firewall rules:

- Add the IP address, and not the host name, of the host on which the JDBC applet is running.
- Ensure that the host name, "`__jdbc__`", never appears in the firewall rules. This host name has been hard-coded as a false host name inside the driver to force an IP address lookup. If you do enter this host name in the list of rules, then every applet using the JDBC Thin driver will be able to go through your firewall.

Writing a URL to Connect Through a Firewall

To write a URL that enables you to connect through a firewall, you must specify the name of the firewall host and the name of the database host to which you want to connect.

For example, if you want to connect to a database on host `oraHost`, listening on port `5221`, with service name `orcl`, and you are going though a firewall on host `fireWallHost`, listening on port `1610`, then use the following URL:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:" +
      "@(description=(address_list=" +
      (address=(protocol=tcp)(host=<firewall-host>)(port=1610))" +
      "(address=(protocol=tcp)(host=oraHost)(port=5221)))" +
      "(source_route=yes)" +
      "(connect_data=(service_name=orcl)))");
);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

Note: To connect through a firewall, you cannot specify the URL in `host:port:service_name` syntax. For example, a URL specified as follows will *not* work:

```
String connString =
   "jdbc:oracle:thin:@example.us.oracle.com:5221:orcl";

OracleDataSource ods = new OracleDataSource();
ods.setURL(connString);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

The first element in the `address_list` represents the connection to the firewall. The second element represents the database to which you want to connect. Note that the order in which you specify the addresses is important.

You can also write the preceding URL in the following format:

```
String connString =
      "jdbc:oracle:thin:@(description=(address_list=
      (address=(protocol=tcp)(port=1600)(host=fireWallHost))
      (address=(protocol=tcp)(port=5221)(host=oraHost)))
      (connect_data=(INSTANCE_NAME=orcl))
      (source_route=yes))";
OracleDataSource ods = new OracleDataSource();
ods.setURL(connString);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

When your applet uses a URL similar to the preceding URL, it will act as if it were connected to the database on host `oraHost`.

Note: All the parameters shown in the preceding example are required. In `address_list`, the firewall address must precede the database server address.

Packaging Applets

After you have coded your applet, you must package it and make it available to users. To package an applet, you will need your applet class files and the JDBC driver class files contained in the `ojdbc6.jar` or `ojdbc7.jar` files.

Follow these steps:

1. Move the JDBC driver classes file `ojdbc6.jar` or `ojdbc7.jar` to an empty directory.

   If your applet connects to a database with a non-`US7ASCII` and non-`WE8ISO8859P1` character set and uses Oracle object types, then also move the `orai18n.jar` file to the same directory.

2. Add your applet classes files to the directory and any other files that the applet may require.

3. Zip the applet classes and driver classes together into a single ZIP or Java Archive (JAR) file. The single ZIP file should contain the following:

- Class files from the `ojdbc6.jar` or `ojdbc7.jar` files and required class files from the `orai18n.jar` files, if the applet requires Globalization Support
- Your applet classes

4. Ensure that the ZIP or JAR file is *not* compressed.

You can now make the applet available to users. One way to do this is to add the `APPLET` tag to the HTML page from which the applet will be run. For example:

```
<APPLET WIDTH=500 HEIGHT=200 CODE=JdbcApplet ARCHIVE=JdbcApplet.zip
      CODEBASE=Applet_Samples
</APPLET>
```

Specifying an Applet in an HTML Page

The `APPLET` tag specifies an applet that runs in the context of an HTML page. The `APPLET` tag can have the following attributes: `CODE`, `ARCHIVE`, `CODEBASE`, `WIDTH`, and `HEIGHT`. These attributes are described in the following sections:

- CODE, HEIGHT, and WIDTH
- CODEBASE
- ARCHIVE

CODE, HEIGHT, and WIDTH

The HTML page that runs the applet must have an `APPLET` tag with an initial width and height to specify the size of the applet display area. You use the `HEIGHT` and `WIDTH` attributes to specify the size, measured in pixels. This size should not count any windows or dialog boxes that the applet opens.

The `APPLET` tag must also specify the name of the file that contains the compiled applet. Specify the file name with the `CODE` attribute. Any path specified must be relative to the base URL of the applet. The path cannot be absolute.

In the following example, `JdbcApplet.class` is the name of the compiled applet:

```
<APPLET CODE="JdbcApplet" WIDTH=500 HEIGHT=200>
</APPLET>
```

If you use this form of the `CODE` attribute, then the classes for the applet and the JDBC Thin driver must be in the same directory as the HTML page.

> **Note:** Do not include the file name extension, `.class`, in the `CODE` attribute.

CODEBASE

The `CODEBASE` attribute is optional. It specifies the base URL of the applet, that is, the name of the directory that contains the code of the applet. If it is not specified, then the URL of the document is used. This means that the classes for the applet and the JDBC Thin driver must be in the same directory as the HTML page. For example, if the current directory is `my_Dir`:

```
<APPLET WIDTH=500 HEIGHT=200 CODE=JdbcApplet CODEBASE="."
</APPLET>
```

The attribute, `CODEBASE="."`, indicates that the applet resides in the current directory, `my_Dir`.

Now, consider that the value of `CODEBASE` is set to `Applet_Samples`, as follows:

```
<APPLET WIDTH=500 HEIGHT=200 CODE=JdbcApplet CODEBASE="Applet_Samples"
</APPLET>
```

This would indicate that the applet resides in the `my_Dir/Applet_Samples` directory.

ARCHIVE

The `ARCHIVE` attribute is optional. It specifies the name of the archive file that contains the applet classes and resources the applet needs. Oracle recommends using an archive file, which saves many extra round-trips to the server.

The archive file will be preloaded. If you have more than one archive file in the list, separate them with commas. In the following example, the class files are stored in the archive file, `JdbcApplet.zip`:

```
<APPLET CODE="JdbcApplet" ARCHIVE="JdbcApplet.zip" WIDTH=500 HEIGHT=200>
</APPLET>
```

Note: Version 3.0 browsers do not support the `ARCHIVE` attribute.

6

Features Specific to JDBC OCI Driver

This chapter introduces the features specific to the Java Database Connectivity (JDBC) Oracle Call Interface (OCI) driver. It also describes the OCI Instant Client. This chapter contains the following sections:

- OCI Connection Pooling
- Client Result Cache
- Transparent Application Failover
- OCI Native XA
- OCI Instant Client
- Instant Client Light (English)

OCI Connection Pooling

The OCI connection pooling feature is an Oracle-designed extension. The connection pooling provided by the JDBC OCI driver enables applications to have multiple logical connections, all of which are using a small set of physical connections. Each call on a logical connection is routed on to the physical connection that is available at the given time.

See Also: Chapter 22, "OCI Connection Pooling"

Client Result Cache

Client result cache feature enables client-side caching of SQL query result sets in client memory. In this way, OCI applications can use client memory to take advantage of the client result cache to improve response times of repetitive queries.

See Also: *Oracle Call Interface Programmer's Guide*

This section covers the following topics:

- Benefits of Client Result Cache
- Usage Guidelines in JDBC

Benefits of Client Result Cache

The benefits of the OCI client-side result set cache are the following:

- The JDBC OCI client-side result set cache is completely transparent to OCI applications and its cache of result set data is kept consistent with any session or database changes that affect its result set.
- Table annotation makes client-side result set work transparently to the JDBC applications. Otherwise, you must use a hint to enable it. The cache hit avoids the execution of the query and roundtrip to the server to get the result sets. This can result in huge performance savings for server resources, for example, server CPU and server I/O.

  See Also: Table Annotations on page 6-2 and SQL Hints on page 6-3

- The result cache on JDBC OCI client is per-process, so multiple client sessions can simultaneously use matching cached result sets.
- The result cache on JDBC OCI client minimizes the need for each OCI application to have its own custom result set cache.
- The result cache on JDBC OCI client uses OCI client memory that is cheaper than server memory.

Usage Guidelines in JDBC

You can enable result caching in the following two ways:

- RESULT_CACHE_MODE Parameter
- Table Annotations
- SQL Hints

Note:

- You must use JDBC statement caching or cache statements at the application level when using the JDBC OCI client result cache. For more information on JDBC statement caching, refer to "Statement and Result Set Caching".
- The SQL hints take precedence over the session parameter `RESULT_CACHE_MODE` and table annotations. The table annotation `FORCE` takes precedence over session parameter.

RESULT_CACHE_MODE Parameter

You can use the `RESULT_CACHE_MODE` parameter to decide the result cache mode across tables in your queries. Use this clause with the `ALTER SESSION` and `ALTER SYSTEM` statements, or inside the server parameter file (`init.ora`) to determine result caching. You can set the `RESULT_CACHE_MODE` parameter to control whether the SQL query result cache is used for all queries, or only for the queries that are annotated with the result cache hint using SQL hints or table annotations.

Table Annotations

You can use table annotations to enable result caching without making changes to the code. The `ALTER TABLE` and `CREATE TABLE` statements enable you to annotate tables with result cache mode. The syntax is:

```
CREATE|ALTER TABLE [<schema>.]<table> ... [RESULT_CACHE (MODE {FORCE|DEFAULT})]
```

Following example shows how to use table annotations with `CREATE TABLE` statements:

```
CREATE TABLE foo (a NUMBER, b VARCHAR2(20)) RESULT_CACHE (MODE FORCE);
```

Following example shows how to use table annotations with `ALTER TABLE` statements:

```
ALTER TABLE foo RESULT_CACHE (MODE DEFAULT);
```

SQL Hints

You can use SQL hints to specify the queries to be cached by annotating the queries with a `/*+ result_cache */` or /*+ no_result_cache */ hint. For example, look at the following code snippet:

```
String  query  = "select /*+ result_cache */ * from employees where employee_id <
: 1";
   ((oracle.jdbc.OracleConnection)conn).setImplicitCachingEnabled(true);
   ((oracle.jdbc.OracleConnection)conn).setStatementCacheSize(10);
   PreparedStatement  pstmt;
   ResultSet rs;

    for (int j = 0 ; j < 10 ; j++ )
    {
       pstmt  = conn.prepareStatement (query);
       pstmt.setInt(1,7500);
       rs  = pstmt.executeQuery();
         while (rs.next( ) )
        {     // see the values  }
          rs.close;
          pstmt.close( ) ;
        }
    }
```

In the preceding example, the client result cache hint `/*+ result_cache */` is annotated to the actual query, that is, `select * from employees where employee_id < : 1`. So, the first execution of the query goes to the database and the result set is cached for the remaining nine executions of the query. This improves the performance of your application significantly. This is primarily useful for read-only data.

Following are some more examples of SQL hints. All the following examples assume that the `dept` table is annotated for result caching by using the following command:

```
ALTER TABLE dept result_cache (MODE FORCE);
```

Examples

- SELECT * FROM employees

  The result set will not be cached.
- SELECT * FROM departments

  The result set will be cached.
- SELECT /*+ result_cache */ employee_id FROM employees

  The result set will be cached.
- SELECT /*+ no_result_cache */ department_id FROM departments

The result set will not be cached.

- SELECT /*+ result_cache */ * FROM departments

  The result set will be cached though query hint is not necessary.

- SELECT e.first_name FROM employees e, departments d WHERE e.department_id = d.department_id

  The result set will not be cached because neither is a query hint available nor are all the tables annotated as `FORCE`.

> **Note:** For information about usage guidelines, Client cache consistency, Deployment Time settings, Client cache Statistics, Validation of client result cache, and OCI Client Result Cache and Server Result Cache, refer to the *Oracle Call Interface Programmer's Guide*.

Transparent Application Failover

The Transparent Application Failover feature of JDBC OCI driver enables you to automatically reconnect to a database if the database instance to which the connection is made goes down. The new database connection, though created by a different node, is identical to the original.

> **See Also:** Chapter 28, "Transparent Application Failover"

OCI Native XA

The JDBC OCI driver also provides a feature called Native XA. This feature enables to use native APIs to send XA commands. The native APIs provide high performance gains as compared to non-native APIs.

> **See Also:** "OCI Native XA" on page 30-20

OCI Instant Client

This section covers the following topics:

Overview of Instant Client

The Instant Client is packaged in a way that makes it extremely easy to deploy OCI, Oracle C++ Call Interface (OCCI), Open Database Connectivity (ODBC), and JDBC-OCI based customer applications, by eliminating the need for an Oracle home. The storage space requirement of a JDBC OCI application using the Instant Client is

significantly reduced compared to the same application running on a full client-side installation. The Instant Client shared libraries occupy only about one-fourth the disk space used by a full client installation.

Table 6–1 shows the Oracle client-side files required to deploy a JDBC OCI application. Library names of Oracle Database 12*c* Release 1 (12.1) are used in the table. The number part of library names will change in future releases to agree with the release.

Table 6–1 OCI Instant Client Shared Libraries

Linux and UNIX Systems	Description for Linux and UNIX Systems	Microsoft Windows	Description for Microsoft Windows
`libclntsh.so.12.1` `libclntshcore.so.12.1`[1]	Client Code Library	`oci.dll`	Forwarding functions that applications link with
`libociei.so`[2]	OCI Instant Client Data Shared Library	`oraociei12.dll`	Data and code
`libnnz12.so`	Security Library	`orannzsbb12.dll`	Security Library
`libocijdbc12.so`	OCI Instant Client JDBC Library	`ocijdbc12.dll`	OCI Instant Client JDBC Library
ALL JDBC Java Archive (JAR) files	**See Also:** "Check the Environment Variables" on page 2-3	All JDBC JAR files	**See Also:** "Check the Environment Variables" on page 2-3

[1] Beginning with Oracle Database 12*c* Release 1, the `libclntshcore.so.12.1` library is separated from the `libclntsh.so.12.1` library and the data shared library.

[2] The `libclntsh.so.12.1` library, the `libclntshcore.so.12.1` library, and the `libociei.so` library must reside in the same directory in order to operate in instant client mode.

> **Note:** To provide Native XA functionality, you must copy the JDBC XA class library. On UNIX systems, this library, `libheteroxa12.so`, is located in the *ORACLE_HOME*`/jdbc/lib` directory. On Microsoft Windows, this library, `heteroxa12.dll`, is located in the *ORACLE_HOME*`\bin` directory.

Benefits of Instant Client

The benefits of Instant Client are the following:

- Installation involves copying a smaller number of files.
- The number of required files and the total disk storage on the Oracle client-side are significantly reduced.
- There is no loss of functionality or performance for applications deployed with the Instant Client.
- It is simple for independent software vendors to package applications.

JDBC OCI Instant Client Installation Process

The Instant Client libraries can be installed by choosing the Instant Client option from Oracle Universal Installer. The Instant Client libraries can also be downloaded from the Oracle Technology Network Web site. The installation process is as follows:

1. Download and install the Instant Client shared libraries and Oracle JDBC class libraries to a directory, such as `instantclient`.
2. Set the library path environment variable to the directory from Step 1. For example, on UNIX systems, set the `LD_LIBRARY_PATH` environment variable to

instantclient. On Microsoft Windows, set the PATH environment variable to locate the instantclient directory.

3. Add the full path names of the JDBC class libraries to the CLASSPATH environment variable.

After completing these steps you are ready to run the JDBC OCI application.

When you use the Instant Client, the OCI and JDBC shared libraries are accessible through the library path environment variable for the JDBC OCI applications. In this case, there is no dependency on the ORACLE_HOME and none of the other code and data files provided in ORACLE_HOME is needed by JDBC OCI, except for the tnsnames.ora file.

Instant Client can be also installed from Oracle Universal Installer by selecting the Instant Client option. The Instant Client files should always be installed in an empty directory. As with the OTN installation, you must set the LD_LIBRARY_PATH environment variable to the Instant Client directory to use the Instant Client.

If you have done a complete client installation by choosing the Admin option, then the Instant Client shared libraries are also installed. The location of the Instant Client shared libraries and JDBC class libraries in a full client installation is:

On Linux or UNIX systems:

- libociei.so library is in $ORACLE_HOME/instantclient
- libclnstsh.so.12.1, libocijdbc12.so, and libnnz12.so are in $ORACLE_HOME/lib
- The JDBC class libraries are in $ORACLE_HOME/jdbc/lib

On Microsoft Windows:

- oraociei12.dll library is in ORACLE_HOME\instantclient
- oci.dll, ocijdbc12.dll, and orannzsbb12.dll are in ORACLE_HOME\bin
- The JDBC class libraries are in ORACLE_HOME\jdbc\lib

By copying these files to a different directory, setting the library path to locate this directory, and adding the path names of the JDBC class libraries to the CLASSPATH environment variable, you can enable running the JDBC OCI application to use the Instant Client.

Note:

- To provide Native XA functionality, you must copy the JDBC XA class library. On UNIX, this library, `libheteroxa12.so`, is located in *ORACLE_HOME*`/jdbc/lib`. On Windows, this library, `heteroxa12.dll`, is located in *ORACLE_HOME*`\bin`.
- All the libraries must be copied from the same *ORACLE_HOME* and must be placed in the same directory.
- On hybrid platforms, such as Sparc64, if the JDBC OCI driver needs to use the Instant Client libraries, then you must copy the `libociei.so` library from the *ORACLE_HOME*`/instantclient32` directory. You must copy all other Sparc64 libraries needed for the JDBC OCI Instant Client from the *ORACLE_HOME*`/lib32` directory.
- Only one set of Oracle libraries should be specified in the library path environment variable. That is, if you have multiple directories containing Instant Client libraries, then only one such directory should be specified in the library path environment variable.
- If you have an Oracle home on your computer, then you should not have the *ORACLE_HOME*`/lib` and Instant Client directories in the library path environment variable simultaneously, regardless of the order in which they appear in the variable. That is, only one of *ORACLE_HOME*`/lib` directory (for non-Instant Client operation) or Instant Client directory (for Instant Client operation) should be specified in the library path environment variable.
- Oracle recommends that you download Instant Client from Oracle Technology Network (OTN)

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

Usage of Instant Client

Instant Client is a deployment feature and should be used for running production applications. For development, a full installation is necessary to access demonstration programs and so on. In general, all JDBC OCI functionality is available to an application using the Instant Client, except that the Instant Client is for client-side operation only. Therefore, server-side external procedures cannot use the Instant Client.

Patching Instant Client Shared Libraries

The Instant Client is a deployment feature, so the emphasis is on reducing the number and size of files required to run a JDBC OCI application. Therefore, all files needed to patch Instant Client shared libraries are not available in an Instant Client deployment. An *ORACLE_HOME* based full client installation is needed to patch the Instant Client shared libraries. The `opatch` utility will take care of patching the Instant Client shared libraries.

Note: On Microsoft Windows, you *cannot* patch the shared libraries.

After applying the patch in an *ORACLE_HOME* environment, copy the files listed in Table 6–1, " OCI Instant Client Shared Libraries" to the instant client directory as described in "JDBC OCI Instant Client Installation Process".

Instead of copying individual files, you can generate Instant Client ZIP files for OCI, OCCI, JDBC, and SQL*Plus as described in "Regeneration of Data Shared Library and ZIP files". Then, you can copy the ZIP files to the target computer and unzip them as described in "JDBC OCI Instant Client Installation Process".

The `opatch` utility stores the patching information of the *ORACLE_HOME* installation in `libclnstsh.so.12.1`. This information can be retrieved by the following command:

```
genezi -v
```

Note that if the computer from where Instant Client is deployed does not have the `genezi` utility, then it must be copied from the *ORACLE_HOME*`/bin` directory on the computer that has the *ORACLE_HOME* installation.

Regeneration of Data Shared Library and ZIP files

The OCI Instant Client Data Shared Library, `libociei.so`, can be regenerated by performing the following steps in an Administrator Installation of *ORACLE_HOME*:

```
mkdir -p $ORACLE_HOME/rdbms/install/instantclient/light
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ilibociei
```

A new version of the `libociei.so` Data Shared Library based on the current files in the *ORACLE_HOME* is then placed in the *ORACLE_HOME*`/rdbms/install/instantclient` directory.

Note that the location of the regenerated Data Shared Library, `libociei.so`, is different from that of the original Data Shared Library, `libociei.so`, which is located in the *ORACLE_HOME*`/instantclient` directory.

The preceding steps also generate Instant Client ZIP files for OCI, OCCI, JDBC, and SQL*Plus.

Regeneration of data shared library and ZIP files is not available on Microsoft Windows platforms.

Database Connection Names for OCI Instant Client

All Oracle Net naming methods that do not require the *ORACLE_HOME* or `TNS_ADMIN` environment variables to locate configuration files, such as `tnsnames.ora` or `sqlnet.ora`, use the Instant Client. In particular, the connection string can be specified in the following formats:

- A Thin-style connection string of the form:

  host`:`*port*`:`*service_name*

  For example:

  ```
  url="jdbc:oracle:oci:@example.com:5521:orcl"
  ```

- A SQL connection URL string of the form:

  `//`*host*`:[`*port*`][/`*service_name*`]`

  For example:

```
url="jdbc:oracle:oci:@//example.com:5521/orcl"
```

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

```
url="jdbc:oracle:oci:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
  (HOST=localhost) (PORT=5521))
  (CONNECT_DATA=(SERVICE_NAME=orcl)))"
```

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

> **See Also:** *Oracle Database Net Services Administrator's Guide* for more information about connection formats

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

> **Note:** In this case, the ORACLE_HOME environment variable is used only for locating Oracle Net configuration files. No other component of Client Code Library uses the value of the ORACLE_HOME environment variable.

The empty connection string is not supported. However, an alternate way to use the empty connection string is to set the TWO_TASK environment variable on UNIX systems, or the LOCAL variable on Microsoft 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 loaded by the TNS_ADMIN or ORACLE_HOME setting.

Example

Consider that the listener.ora file on the database server contains the following information:

```
LISTENER = (ADDRESS_LIST=(ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=5221)))

SID_LIST_LISTENER = (SID_LIST=
                     (SID_DESC=(SID_NAME=rdbms3)
                     (GLOBAL_DBNAME=rdbms3.server6.com)
                     (ORACLE_HOME=/home/dba/rdbms3/oracle)))
```

You can connect to this server in one of the following ways:

```
url = "jdbc:oracle:oci:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
                     (HOST=server6)(PORT=5221))
                     (CONNECT_DATA=(SERVICE_NAME=rdbms3.server6.com)))"
```

or:

```
url = "jdbc:oracle:oci:@//server6:5221/rdbms3.server6.com"
```

Alternatively, you can set the TWO_TASK environment variable to any of the connection strings and connect to the database server without specifying the connection string along with the sqlplus command. For example, set the TWO_TASK environment in one of the following ways:

```
setenv TWO_TASK "(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=5221))
                 (CONNECT_DATA=(SERVICE_NAME=rdbms3.server6.com)))"
```

or:

```
setenv TWO_TASK //server6:5221/rdbms3.server6.com
```

Now, you can connect to the database server using the following URL:

```
url = "jdbc:oracle:oci:@"
```

The connection string can also be stored in the tnsnames.ora file. For example, consider that the tnsnames.ora file contains the following:

```
conn_str = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=5221))
                (CONNECT_DATA=(SERVICE_NAME=rdbms3.server6.com)))
```

If this tnsnames.ora file is located in the /home/webuser/instantclient directory, then you can set the TNS_ADMIN environment variable (or LOCAL on Microsoft Windows) as follows:

```
setenv TNS_ADMIN /home/webuser/instantclient
```

Now, you can connect as follows:

```
url = "jdbc:oracle:oci:@conn_str"
```

Note: The TNS_ADMIN environment variable specifies the directory where the tnsnames.ora file is located. However, TNS_ADMIN does not specify the full path of the tnsnames.ora file, instead it specifies the directory.

If this tnsnames.ora file is located in the /network/server6/home/dba/oracle/network/admin directory in the Oracle home, then instead of using TNS_ADMIN to locate the tnsnames.ora file, you can set the ORACLE_HOME environment variable as follows:

```
setenv ORACLE_HOME /network/server6/home/dba/oracle
```

Now, you can connect with either of the conn_str connection strings, as specified previously.

If tnsnames.ora can be located by TNS_ADMIN or ORACLE_HOME, then TWO_TASK can be set to:

```
setenv TWO_TASK conn_str
```

You can then connect with the following URL:

```
url = "jdbc:oracle:oci:@"
```

Environment Variables for OCI Instant Client

The ORACLE_HOME environment variable no longer determines the location of the NLS, CORE, and error message files. An OCI-only application does not require the ORACLE_HOME environment variable to be set. However, if the variable is set, then it does not have an impact on the operation of the OCI driver. OCI driver always obtains its data from the Data Shared Library. If the Data Shared Library is not available, only then the ORACLE_HOME environment variable is used and a full client installation is assumed. Even though the ORACLE_HOME environment variable is not required to be set, if it is set, then it must be set to a valid operating system path name that identifies a directory.

Environment variables ORA_NLS10 and ORA_NLSPROFILES33 are ignored while using the Instant Client.

If the ORA_TZFILE variable is not set, then the Instant Client uses the larger timezlrg_n.dat file from the Data Shared Library, which is the default setting. If the smaller timezone_n.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 UNIX systems:

```
setenv ORA_TZFILE timezone_n.dat
```

On Microsoft Windows:

```
set ORA_TZFILE timezone_n.dat
```

In the examples above, *n* is the time zone data file version number.

If the OCI driver is not using the Instant Client because of nonavailability of the Data Shared Library, then the ORA_TZFILE variable, if set, names a complete path name, as it does in previous Oracle Database releases.

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

Instant Client Light (English)

The lightweight version of Instant Client is called Instant Client Light (English). Instant Client Light is the short name. Instant Client Light is a significantly smaller version of Instant Client. This reduces the disk space requirements of the client installation by about 63 MB. This is achieved by the lightweight Data Shared Library, libociicus.so on UNIX systems, which is 4 MB in size and a subset of the data shared library, libociei.so, which is 67 MB in size.

The lightweight data shared library supports only a few character sets and error messages that are only in English. Therefore, the name Instant Client Light (English). Instant Client Light is designed for applications that require English-only error messages and use either US7ASCII, WE8DEC, or one of the Unicode character sets.

Table 6–2 lists the names of the data shared libraries for Instant Client and Instant Client Light (English) on different platforms. The table also specifies the size of each data shared library in parentheses following the library file name.

Table 6–2 Data Shared Library for Instant Client and Instant Client Light (English)

Platform	Instant Client	Instant Client Light (English)
Solaris	libociei.so (67 MB)	libociicus.so (4 MB)
Linux	libociei.so (67 MB)	libociicus.so (4 MB)
Microsoft Windows	oraociei12.dll (85 MB)	oraociicus12.dll (15 MB)

This section covers the following topics:

- Globalization Settings
- Operation
- Installation

Globalization Settings

The NLS_LANG setting determines the language, territory, and character set as *language_territory*.*characterset*. In Instant Client Light, *language* can only be American, *territory* can be any that is supported, and *characterset* can be any one of the following:

- Single-byte
  - US7ASCII
  - WE8DEC
  - WE8MSWIN1252
  - WE8ISO8859P1
- Unicode
  - UTF8
  - AL16UTF16
  - AL32UTF8

Specifying character set or national character set other than those listed as the client or server character set or setting the language in NLS_LANG on the client will throw one of the following errors:

- ORA-12734
- ORA-12735
- ORA-12736
- ORA-12737

With Instant Client Light, the error messages obtained are only in English. Therefore, the valid values for the NLS_LANG setting are of the type:

```
American_territory.characterset
```

where, *territory* can be any valid and supported territory and *characterset* can be any one the previously listed character sets.

Instant Client Light can operate with the OCI environment handles created in the OCI_UTF16 mode.

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

Operation

To use the Instant Client Light, an application must set the LD_LIBARARY_PATH environment variable in UNIX systems or the PATH environment variable in Microsoft Windows to a location containing the client and data shared libraries. OCI applications by default look for the OCI Data Shared Library, libociei.so in the LD_LIBRARY_PATH environment variable in UNIX systems or the oraociei12.dll Data Shared Library in the PATH environment variable in Microsoft Windows, to determine if the application should use the Instant Client. In case this library is not found, then OCI tries to load the Instant Client Light Data Shared Library, libociicus.so in UNIX systems or libociicus12.dll in Microsoft Windows. If this library is found, then the application uses the Instant Client Light. Otherwise, a non-Instant Client is used.

Installation

Instant Client Light can be installed in one of the following ways:

- From OTN

  You can download the required file from

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

  For Instant Client Light, instead of downloading and expanding the Basic package, download and unzip the Basic Light package. The directory in which the lightweight libraries are unzipped should be empty before unzipping the files.

- From Client Admin Install

  Instead of copying `libociei.so` or `oraociei12.dll` from the *ORACLE_HOME*`/instantclient` directory, copy `libociicus.so` or `oraociic12.dll` from the *ORACLE_HOME*`/instantclient/light` directory. That is, the Instant Client directory on the `LD_LIBRARY_PATH` environment variable, in UNIX systems, should contain the Instant Client Light Data Shared Library, `libociicus.so`, instead of the larger OCI Instant Client Data Shared Library, `libociei.so`. In Microsoft Windows, the `PATH` environment variable should contain `oraociicus12.dll` instead of `oraociei12.dll`.

- From Oracle Universal Installer

  If the Instant Client option is selected from Oracle Universal Installer, then `libociei.so` (or `oraociei12.dll` on Microsoft Windows) is installed in the base directory of the installation which is going to be placed on the `LD_LIBRARY_PATH` environment variable. This is so that Instant Client Light is not enabled by default. The Instant Client Light Data Shared Library, `libociicus.so` (or `oraociicus12.dll` on Microsoft Windows), is installed in the `light` subdirectory of the base directory. Therefore, to use in the Instant Client Light, the OCI Data Shared Library, `libociei.so` (or `oraociei12.dll` on Windows) must be deleted or renamed and the Instant Client Light Data Shared Library must be copied from the `light` subdirectory to the base directory of the installation.

  For example, if Oracle Universal Installer has installed the Instant Client in `my_oraic_12_1` directory on the `LD_LIBRARY_PATH` environment variable, then you must perform the following to use the Instant Client Light:

```
cd my_oraic_12_1
rm libociei.so
mv light/libociicus.so .
```

> **Note:** All the Instant Client files should always be copied or installed in an empty directory. This is to ensure that no incompatible binaries exist in the installation.

7

Server-Side Internal Driver

This chapter covers the following topics:

Overview of the Server-Side Internal Driver

The server-side internal driver is intrinsically tied to Oracle Database and to the embedded Java Virtual Machine, also known as Oracle Java Virtual Machine (Oracle JVM). The driver runs as part of the same process as the Database. It also runs within the default session, the same session in which the Oracle JVM was started. Each Oracle JVM session has a single implicit native connection to the Database session in which it exists. This connection is conceptual and is not a Java object. It is an inherent aspect of the session and cannot be opened or closed from within the JVM.

The server-side internal driver is optimized to run within the database server and provide direct access to SQL data and PL/SQL subprograms on the local database. The entire JVM operates in the same address space as the database and the SQL engine. Access to the SQL engine is a function call. This enhances the performance of your Java Database Connectivity (JDBC) applications and is much faster than running a remote Oracle Net call to access the SQL engine.

The server-side internal driver supports the same features, application programming interfaces (APIs), and Oracle extensions as the client-side drivers. This makes application partitioning very straightforward. For example, if you have a Java application that is data-intensive, then you can easily move it into the database server for better performance, without having to modify the application-specific calls.

Connecting to the Database

As described in the preceding section, the server-side internal driver runs within a default session. Therefore, you are already connected. There are two methods to access the default connection:

- Use the `OracleDataSource.getConnection` method, with any of the following forms as the URL string:
  - `jdbc:oracle:kprb`

  - `jdbc:default:connection`
  - `jdbc:oracle:kprb:`
  - `jdbc:default:connection:`
- Use the Oracle-specific `defaultConnection` method of the `OracleDriver` class.

Using `defaultConnection` is generally recommended.

> **Note:** You are no longer required to register the `OracleDriver` class for connecting with the server-side internal driver.

Connecting with the OracleDriver Class defaultConnection Method

The `defaultConnection` method of the `oracle.jdbc.OracleDriver` class is an Oracle extension and always returns the same connection object. Even if you call this method multiple times, assigning the resulting connection object to different variable names, then only a single connection object is reused.

You need *not* include a connection string in the `defaultConnection` call. For example:

```
import java.sql.*;
import oracle.jdbc.*;

class JDBCConnection
{
  public static Connection connect() throws SQLException
  {
    Connection conn = null;
    try {
      // connect with the server-side internal driver
         conn = ora.defaultConnection();
      }

    } catch (SQLException e) {...}
    return conn;
  }
}
```

Note that there is no `conn.close` call in the example. When JDBC code is running inside the target server, the connection is an implicit data channel, not an explicit connection instance as from a client. It should *not* be closed.

OracleDriver has a static variable to store a default connection instance. The method `OracleDriver.defaultConnection` returns this default connection instance if the connection exists and is not closed. Otherwise, it creates a new, open instance and stores it in the static variable and returns it to the caller.

Typically, you should use the `OracleDriver.defaultConnection` method. This method is faster and uses less resources. Java stored procedures should be carefully written. For example, to close statements before the end of each call.

Typically, you should not close the default connection instance because it is a single instance that can be stored in multiple places, and if you close the instance, each would become unusable. If it is closed, a later call to the `OracleDriver.defaultConnection` method gets a new, open instance.

Connecting with the OracleDataSource.getConnection Method

To connect to the internal server connection from code that is running within the target server, you can use the `OracleDataSource.getConnection` method with either of the following URLs:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:kprb");
Connection conn = ods.getConnection();
```

or:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:default:connection");
Connection conn = ods.getConnection();
```

or:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:kprb:");
Connection conn = ods.getConnection();
```

or:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:default:connection:");
Connection conn = ods.getConnection();
```

Any user name or password you include in the URL is ignored in connecting to the default server connection.

The `OracleDataSource.getConnection` method returns a new Java `Connection` object every time you call it. The fact that `OracleDataSource.getConnection` returns a new connection object every time you call it is significant if you are working with object maps or type maps. A type map is associated with a specific `Connection` object and with any state that is part of the object. If you want to use multiple type maps as part of your program, then you can call `getConnection` to create a new `Connection` object for each type map.

> **Note:** Although the `OracleDataSource.getConnection` method returns a new object every time you call it, it does not create a new database connection every time. They all utilize the same implicit native connection and share the same session state, in particular, the local transaction.

Session and Transaction Context

The server-side driver operates within a default session and default transaction context. The default session is the session in which the JVM was started. In effect, you are already connected to the database on the server. This is different from the client-side where there is no default session. You must explicitly connect to the database.

Auto-commit mode is disabled in the server. You must manage transaction `COMMIT` and `ROLLBACK` operations explicitly by using the appropriate methods on the connection object:

```
conn.commit();
```

or:

```
conn.rollback();
```

Note: As a best practice, it is recommended not to commit or rollback a transaction inside the server.

Testing JDBC on the Server

Almost any JDBC program that can run on a client can also run on the server. All the programs in the `samples` directory can be run on the server, with only minor modifications. Usually, these modifications concern only the connection statement.

Consider the following code fragment which obtains a connection to a database:

```
ods.setUrl(
"jdbc:oracle:oci:@(DESCRIPTION=
  (ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias)
    (PORT=5221))
    (CONNECT_DATA=(SERVICE_NAME=orcl)))");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

We can modify this code fragment for use in the server-side internal driver. In the server-side internal driver, no user, password, or database information is necessary. For the connection statement, you use:

```
ods.setUrl(
"jdbc:oracle:kprb:@");
Connection conn = ods.getConnection();
```

However, the most convenient way to get a connection is to call the `OracleDriver.defaultConnection` method, as follows:

```
Connection conn = OracleDriver.defaultConnection();
```

Loading an Application into the Server

When loading an application into the server, you can load `.class` files that you have already compiled on the client or you can load `.java` source files and have them automatically compiled on the server.

Using the Loadjava Utility

You can use the `loadjava` utility to load your files. You can either specify source file names on the command line or put the files into a Java Archive (JAR) file and specify the JAR file name on the command line.

The `loadjava` script, which runs the actual utility, is in the `bin` directory in your Oracle home. This directory should already be in your path once Oracle has been installed.

Note: The `loadjava` utility supports compressed files.

Loading Class Files into the Server

Consider a case where you have the following three class files in your application: `Foo1.class`, `Foo2.class`, and `Foo3.class`. Each class is written into its own class schema object in the server.

You can load the class files using the default JDBC Oracle Call Interface (OCI) driver in the following ways:

- Specifying the individual class file names, as follows:

```
loadjava -user HR Foo1.class Foo2.class Foo3.class
Password: password
```

- Specifying the class file names using a wildcard, as follows:

```
loadjava -user HR Foo*.class
Password: password
```

- Specifying a JAR file that contains the class files, as follows:

```
loadjava -user HR Foo.jar
Password: password
```

You can load the files using the JDBC Thin driver, as follows:

```
loadjava -thin -user HR@localhost:5221:orcl Foo.jar
Password: password
```

Note: Starting from Oracle Database 12*c* Release 1 (12.1), JDK 6, and JDK 7 are supported. However, only one of the JVMs will be active at a given time.

Ensure that your classes are not compiled using a newer version of JDK than the active runtime version on the server.

Loading Source Files into the Server

If you enable the `loadjava -resolve` option when loading a `.java` source file, then the server-side compiler will compile your application as it is loaded, resulting in both a source schema object for the original source code and one or more class schema objects for the compiled output.

If you do not specify `-resolve`, then the source is loaded into a source schema object without any compilation. In this case, however, the source is implicitly compiled the first time an attempt is made to use a class defined in the source.

For example, run `loadjava` as follows to load and compile `Foo.java`, using the default JDBC OCI driver:

```
loadjava -user HR -resolve Foo.java
Password: password
```

Or, use the following command to load using the JDBC Thin driver:

```
loadjava -thin -user HR@localhost:5221:orcl -resolve Foo.java
Password: password
```

Either of these will result in appropriate class schema objects being created in addition to the source schema object.

Note: Oracle generally recommends compiling source on the client, whenever possible, and loading the `.class` files instead of the source files into the server.

See Also: *Oracle Database Java Developer's Guide*

Using the JVM Command Line

You can also use the JVM command-line option to load your files. The command-line interface to Oracle JVM is analogous to using the JDK or JRE shell commands. You can:

- Use the standard `-classpath` syntax to indicate where to find the classes to load
- Set the system properties by using the standard `-D` syntax

The interface is a PL/SQL function that takes a string (`VARCHAR2`) argument, parses it as a command-line input and if it is properly formed, runs the indicated Java method in Oracle JVM. To do this, PL/SQL package `DBMS_JAVA` provides the following functions:

- `runjava`

  You can use the `runjava` function in the following way:

  ```
  FUNCTION runjava(cmdline VARCHAR2) RETURN VARCHAR2;
  ```

- `runjava_in_current_session`

  You can use the `runjava_in_current_session` function in the following way:

  ```
  FUNCTION runjava_in_current_session(cmdline VARCHAR2) RETURN VARCHAR2;
  ```

Note: Starting with Oracle Database 11*g* Release 1, there is a just-in-time (JIT) compiler for Oracle JVM environment. A JIT compiler for Oracle JVM enables much faster execution because the JIT compiler uses advanced techniques as compared to the old Native compiler and compiles dynamically generated code. Unlike the old Native compiler, the JIT compiler does not require a C compiler. It is enabled without the support of any plug-ins.

For more information, refer to *Oracle Database Java Developer's Guide*.

Part III

Connection and Security

This part consists of chapters that discuss the use of data sources and URLs to connect to the database. It also includes chapters that discuss the security features supported by the Oracle Java Database Connectivity (JDBC) Oracle Call Interface (OCI) and Thin drivers, Secure Sockets Layer (SSL) support in JDBC Thin driver, and middle-tier authentication through proxy connections.

Part III contains the following chapters:

8

Data Sources and URLs

This chapter discusses connecting applications to databases using Java Database Connectivity (JDBC) data sources, as well as the URLs that describe databases. This chapter contains the following sections:

- Data Sources
- Database URLs and Database Specifiers

Data Sources

Data sources are standard, general-use objects for specifying databases or other resources to use. The JDBC 2.0 extension application programming interface (API) introduced the concept of data sources. For convenience and portability, data sources can be bound to Java Naming and Directory Interface (JNDI) entities, so that you can access databases by logical names.

The data source facility provides a complete replacement for the previous JDBC `DriverManager` facility. You can use both facilities in the same application, but it is recommended that you transition your application to data sources.

This section covers the following topics:

- Overview of Oracle Data Source Support for JNDI
- Features and Properties of Data Sources
- Creating a Data Source Instance and Connecting
- Creating a Data Source Instance, Registering with JNDI, and Connecting
- Supported Connection Properties
- Using Roles for SYS Login
- Configuring Database Remote Login
- Bequeath Connection and SYS Logon
- Properties for Oracle Performance Extensions

Overview of Oracle Data Source Support for JNDI

The JNDI standard provides a way for applications to find and access remote services and resources. These services can be any enterprise services. However, for a JDBC application, these services would include database connections and services.

JNDI enables an application to use logical names in accessing these services, removing vendor-specific syntax from application code. JNDI has the functionality to associate a logical name with a particular source for a desired service.

All Oracle JDBC data sources are JNDI-referenceable. The developer is not required to use this functionality, but accessing databases through JNDI logical names makes the code more portable.

> **Note:** Using JNDI functionality requires the `jndi.jar` file to be in the `CLASSPATH` environment variable. This file is included with the Java products on the installation CD. You must add it to the `CLASSPATH` environment variable separately.

Features and Properties of Data Sources

By using the data source functionality with JNDI, you do not need to register the vendor-specific JDBC driver class name and you can use logical names for URLs and other properties. This ensures that the code for opening database connections is portable to other environments.

The DataSource Interface and Oracle Implementation

A JDBC data source is an instance of a class that implements the standard `javax.sql.DataSource` interface:

```
public interface DataSource
{
   Connection getConnection() throws SQLException;
   Connection getConnection(String username, String password)
      throws SQLException;
   ...
}
```

Oracle implements this interface with the `OracleDataSource` class in the `oracle.jdbc.pool` package. The overloaded `getConnection` method returns a connection to the database.

To use other values, you can set properties using appropriate setter methods. For alternative user names and passwords, you can also use the `getConnection` method that takes these parameters as input. This would take priority over the property settings.

> **Note:** The `OracleDataSource` class and all subclasses implement the `java.io.Serializable` and `javax.naming.Referenceable` interfaces.

Properties of DataSource

The `OracleDataSource` class, as with any class that implements the `DataSource` interface, provides a set of properties that can be used to specify a database to connect to. These properties follow the JavaBeans design pattern.

Table 8–1 and Table 8–2 list `OracleDataSource` properties. The properties in Table 8–1 are standard properties. The properties in Table 8–2 are Oracle extensions.

> **Note:** Oracle does not implement the standard `roleName` property.

Table 8–1 Standard Data Source Properties

Name	Type	Description
databaseName	String	Name of the particular database on the server.
dataSourceName	String	Name of the underlying data source class. For connection pooling, this is an underlying pooled connection data source class. For distributed transactions, this is an underlying XA data source class.
description	String	Description of the data source.
networkProtocol	String	Network protocol for communicating with the server. For Oracle, this applies only to the JDBC Oracle Call Interface (OCI) drivers and defaults to tcp.
password	String	Password for the connecting user.
portNumber	int	Number of the port where the server listens for requests
serverName	String	Name of the database server
user	String	Name for the login

Note: For security reasons, there is no getPassword() method.

Table 8–2 Oracle Extended Data Source Properties

Name	Type	Description
connectionCacheName	String	Specifies the name of the cache. This cannot be changed after the cache has been created.
connectionCacheProperties	java.util.Properties	Specifies properties for implicit connection cache.
connectionCachingEnabled	Boolean	Specifies whether implicit connection cache is in use.
connectionProperties	java.util.Properties	Specifies the connection properties.
driverType	String	Specifies Oracle JDBC driver type. It can be one of oci, thin, or kprb.
fastConnectionFailoverEnabled	Boolean	Specifies whether Fast Connection Failover is in use.
implicitCachingEnabled	Boolean	Specifies whether the implicit statement connection cache is enabled.
loginTimeout	int	Specifies the maximum time in seconds that this data source will wait while attempting to connect to a database.
logWriter	java.io.PrintWriter	Specifies the log writer for this data source.
maxStatements	int	Specifies the maximum number of statements in the application cache.
serviceName	String	Specifies the database service name for this data source.

Table 8–2 (Cont.) Oracle Extended Data Source Properties

Name	Type	Description
tnsEntry	String	Specifies the TNS entry name. The TNS entry name corresponds to the TNS entry specified in the tnsnames.ora configuration file. Enable this OracleXADataSource property when using the Native XA feature with the OCI driver, to access Oracle pre-8.1.6 databases and later. If the tnsEntry property is not set when using the Native XA feature, then a SQLException with error code ORA-17207 is thrown
url	String	Specifies the URL of the database connection string. Provided as a convenience, it can help you migrate from an older Oracle Database. You can use this property in place of the Oracle tnsEntry and driverType properties and the standard portNumber, networkProtocol, serverName, and databaseName properties.
nativeXA	Boolean	Allows an OracleXADataSource using the Native XA feature with the OCI driver, to access Oracle pre-8.1.6 databases and later. If the nativeXA property is enabled, be sure to set the tnsEntry property as well. This property is only for OracleXADatasource. This DataSource property defaults to false.
ONSConfiguration	String	Specifies the ONS configuration string that is used to remotely subscribe to FAN/ONS events.

Note:

- This table omits properties that supported the deprecated connection cache based on OracleConnectionCache.
- Because Native XA performs better than Java XA, use Native XA whenever possible.

Use the setConnectionProperties method to set the properties of the connection and the setConnectionCacheProperties method to set the properties of the connection cache.

For more information about the properties of the connection refer to "Supported Connection Properties" on page 8-6.

If you are using the server-side internal driver, that is, the driverType property is set to kprb, then any other property settings are ignored.

If you are using the JDBC Thin or OCI driver, then note the following:

- A URL setting can include settings for user and password, as in the following example, in which case this takes precedence over individual user and password property settings:

  ```
  jdbc:oracle:thin:HR/hr@localhost:5221:orcl
  ```

- Settings for user and password are required, either directly through the URL setting or through the getConnection call. The user and password settings in a getConnection call take precedence over any property settings.

- If the `url` property is set, then any `tnsEntry`, `driverType`, `portNumber`, `networkProtocol`, `serverName`, and `databaseName` property settings are ignored.
- If the `tnsEntry` property is set, which presumes the `url` property is not set, then any `databaseName`, `serverName`, `portNumber`, and `networkProtocol` settings are ignored.
- If you are using an OCI driver, which presumes the `driverType` property is set to `oci`, and the `networkProtocol` is set to `ipc`, then any other property settings are ignored.

Also, note that `getConnectionCacheName()` will return the name of the cache only if the `ConnectionCacheName` property of the data source is set after caching is enabled on the data source.

Creating a Data Source Instance and Connecting

This section shows an example of the most basic use of a data source to connect to a database, without using JNDI functionality. Note that this requires vendor-specific, hard-coded property settings.

Create an `OracleDataSource` instance, initialize its connection properties as appropriate, and get a connection instance, as in the following example:

```
OracleDataSource ods = new OracleDataSource();
ods.setDriverType("oci");
ods.setServerName("localhost");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("816");
ods.setPortNumber(5221);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

Or, optionally, override the user name and password, as follows:

```
Connection conn = ods.getConnection("OE", "oe");
```

Creating a Data Source Instance, Registering with JNDI, and Connecting

This section exhibits JNDI functionality in using data sources to connect to a database. Vendor-specific, hard-coded property settings are required only in the portion of code that binds a data source instance to a JNDI logical name. From that point onward, you can create portable code by using the logical name in creating data sources from which you will get your connection instances.

Note: Creating and registering data sources is typically handled by a JNDI administrator, not in a JDBC application.

Initialize Data Source Properties

Create an `OracleDataSource` instance, and then initialize its properties as appropriate, as in the following example:

```
OracleDataSource ods = new OracleDataSource();
ods.setDriverType("oci");
ods.setServerName("localhost");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("816");
```

```
ods.setPortNumber(5221);
ods.setUser("HR");
ods.setPassword("hr");
```

Register the Data Source

Once you have initialized the connection properties of the `OracleDataSource` instance `ods`, as shown in the preceding example, you can register this data source instance with JNDI, as in the following example:

```
Context ctx = new InitialContext();
ctx.bind("jdbc/sampledb", ods);
```

Calling the JNDI `InitialContext()` constructor creates a Java object that references the initial JNDI naming context. System properties, which are not shown, instruct JNDI which service provider to use.

The `ctx.bind` call binds the `OracleDataSource` instance to a logical JNDI name. This means that anytime after the `ctx.bind` call, you can use the logical name `jdbc/sampledb` in opening a connection to the database described by the properties of the `OracleDataSource` instance `ods`. The logical name `jdbc/sampledb` is logically bound to this database.

The JNDI namespace has a hierarchy similar to that of a file system. In this example, the JNDI name specifies the subcontext `jdbc` under the root naming context and specifies the logical name `sampledb` within the `jdbc` subcontext.

The `Context` interface and `InitialContext` class are in the standard `javax.naming` package.

> **Note:** The JDBC 2.0 Specification requires that all JDBC data sources be registered in the `jdbc` naming subcontext of a JNDI namespace or in a child subcontext of the `jdbc` subcontext.

Open a Connection

To perform a lookup and open a connection to the database logically bound to the JNDI name, use the logical JNDI name. Doing this requires casting the lookup result, which is otherwise a Java `Object`, to `OracleDataSource` and then using its `getConnection` method to open the connection.

Here is an example:

```
OracleDataSource odsconn = (OracleDataSource)ctx.lookup("jdbc/sampledb");
Connection conn = odsconn.getConnection();
```

Supported Connection Properties

For a detailed list of connection properties that Oracle JDBC drivers support, see the *Oracle Database JDBC Java API Reference*.

Using Roles for SYS Login

To specify the role for the `SYS` login, use the `internal_logon` connection property. To log on as `SYS`, set the `internal_logon` connection property to `SYSDBA` or `SYSOPER`.

For a bequeath connection, we can get a connection as SYS by setting the internal_logon property. For a remote connection, we need additional password file setting procedures.

Configuring Database Remote Login

Before the JDBC Thin driver can connect to the database as SYSDBA, you must configure the user, because Oracle Database security system requires a password file for remote connections as an administrator. Perform the following:

1. Set a password file on the server-side or on the remote database, using the orapwd password utility. You can add a password file for user SYS as follows:

   - In UNIX

     ```
     orapwd file=$ORACLE_HOME/dbs/orapwORACLE_SID entries=200
     Enter password: password
     ```

   - In Microsoft Windows

     ```
     orapwd file=%ORACLE_HOME%\database\PWDORACLE_SID.ora entries=200
     Enter password: password
     ```

   In this case, file is the name of the password file, *password* is the password for user SYS. It can be altered using the ALTER USER statement in SQL Plus. You should set entries to a value higher than the number of entries you expect.

   The syntax for the password file name is different on Microsoft Windows and UNIX.

   See Also: *Oracle Database Administrator's Guide*

2. Enable remote login as SYSDBA. This step grants SYSDBA and SYSOPER system privileges to individual users and lets them connect as themselves.

   Stop the database, and add the following line to init*service_name*.ora, in UNIX, or init.ora, in Microsoft Windows:

   ```
   remote_login_passwordfile=exclusive
   ```

   The init*service_name*.ora file is located at *ORACLE_HOME*/dbs/ and also at *ORACLE_HOME*/admin/db_name/pfile/. Ensure that you keep the two files synchronized.

   The init.ora file is located at %ORACLE_BASE%\ADMIN\db_name\pfile\.

3. Change the password for the SYS user. This is an optional step.

   ```
   PASSWORD sys
         Changing password for sys
   New password: password
   Retype new password: password
   ```

4. Verify whether SYS has the SYSDBA privilege.

   ```
   SQL> select * from v$pwfile_users;
   USERNAME                       SYSDB       SYSOP
   ----------------------         ---------   ---------
   SYS                            TRUE        TRUE
   ```

5. Restart the remote database.

Example 8–1 Using SYS Login To Make a Remote Connection

```
//This example works regardless of language settings of the database.
 /** case of remote connection using sys **/
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
// create an OracleDataSource
OracleDataSource ods = new OracleDataSource();
// set connection properties
java.util.Properties prop = new java.util.Properties();
prop.put("user", "sys");
prop.put("password", "sys");
prop.put("internal_logon", "sysoper");
ods.setConnectionProperties(prop);
// set the url
// the url can use oci driver as well as:
// url = "jdbc:oracle:oci8:@remotehost"; the remotehost is a remote database
String url = "jdbc:oracle:thin:@//localhost:5221/orcl";
ods.setURL(url);
// get the connection
Connection conn = ods.getConnection();
...
```

Bequeath Connection and SYS Logon

The following example illustrates how to use the `internal_logon` and `SYSDBA` arguments to specify the `SYS` login. This example works regardless of the database's national-language settings of the database.

```
/** Example of bequeath connection **/
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;

// create an OracleDataSource instance
OracleDataSource ods = new OracleDataSource();

// set neccessary properties
java.util.Properties prop = new java.util.Properties();
prop.put("user", "sys");
prop.put("password", "sys");
prop.put("internal_logon", "sysdba");
ods.setConnectionProperties(prop);

// the url for bequeath connection
String url = "jdbc:oracle:oci8:@";
ods.setURL(url);

// retrieve the connection
Connection conn = ods.getConnection();
...
```

Properties for Oracle Performance Extensions

Some of the connection properties are for use with Oracle performance extensions. Setting these properties is equivalent to using corresponding methods on the `OracleConnection` object, as follows:

- Setting the `defaultRowPrefetch` property is equivalent to calling `setDefaultRowPrefetch`.
- Setting the `remarksReporting` property is equivalent to calling `setRemarksReporting`.

  See Also: "Reporting DatabaseMetaData TABLE_REMARKS" on page 21-21

- Setting the `defaultBatchValue` property is equivalent to calling `setDefaultExecuteBatch`

  See Also: "Oracle Update Batching" on page 21-3

Example

The following example shows how to use the `put` method of the `java.util.Properties` class, in this case, to set Oracle performance extension parameters.

```
//import packages and register the driver
import java.sql.*;
import java.math.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;

//specify the properties object
java.util.Properties info = new java.util.Properties();
info.put ("user", "HR");
info.put ("password", "hr");
info.put ("defaultRowPrefetch","20");
info.put ("defaultBatchValue", "5");

//specify the datasource object
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@//localhost:5221/orcl");
ods.setUser("HR");
ods.setPassword("hr");
ods.setConnectionProperties(info);
...
```

Database URLs and Database Specifiers

Database URLs are strings. The complete URL syntax is:

`jdbc:oracle:`*`driver_type`*`:[`*`username`*`/`*`password`*`]@`*`database_specifier`*

Note:

- The brackets indicate that the *`username`*`/`*`password`* pair is optional.
- `kprb`, the internal server-side driver, uses an implicit connection. Database URLs for the server-side driver end after the *`driver_type`*.

The first part of the URL specifies which JDBC driver is to be used. The supported *driver_type* values are `thin`, `oci`, and `kprb`.

The remainder of the URL contains an optional user name and password separated by a slash, an @, and the database specifier, which uniquely identifies the database to which the application is connected. Some database specifiers are valid only for the JDBC Thin driver, some only for the JDBC OCI driver, and some for both.

Support for Internet Protocol Version 6

This release of Oracle JDBC drivers supports Internet Protocol Version 6 (IPv6) addresses in the JDBC URL and machine names that resolve to IPv6 addresses. IPv6 is a new Network layer protocol designed by the Internet Engineering Task Force (IETF) to replace the current version of Internet Protocol, Internet Protocol Version 4 (IPv4). The primary benefit of IPv6 is a large address space, derived from the use of 128-bit addresses. IPv6 also improves upon IPv4 in areas such as routing, network autoconfiguration, security, quality of service, and so on.

Note:

- An IPv6 Client can support only IPv6 Servers or servers with dual protocol support, that is, support for both IPv6 and IPv4 protocols. Conversely, an IPv6 Server can support only IPv6 clients or dual protocol clients.
- IPv6 is supported only with single instance Database servers and not with Oracle RAC.

If you want to use a literal IPv6 address in a URL, then you should enclose the literal address enclosed in a left bracket ([) and a right bracket (]). For example: `[2001:0db8:7654:3210:FEDC:BA98:7654:3210]`. So, a JDBC URL, using an IPv6 address will look like the following:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
  (HOST=[2001:0db8:7654:3210:FEDC:BA98:7654:3210])(PORT=5521))
  (CONNECT_DATA=(SERVICE_NAME=sales.example.com))
```

Note: All the new System classes that are required for IPv6 support are loaded when Java is enabled during database initialization. So, if your application does not have any IPv6 addressing, then you do not need to change your code to use IPv6 functionality. However, if your application has either IPv6 only or both IPv6 and IPv4 addressing, then you should set the java.net.preferIPv6Addresses system property in the command line. This enables the Oracle JVM to load appropriate libraries. These libraries are loaded once and cannot be reloaded without restarting the Java process. For more information about this system property, refer to *Oracle Database Java Developer's Guide*.

Database Specifiers

Table 8–3, shows the possible database specifiers, listing which JDBC drivers support each specifier.

Note:

- Starting Oracle Database 10*g*, Oracle Service IDs are not supported.
- Starting Oracle Database 10*g*, Oracle no longer supports Oracle Names as a naming method.

Table 8–3 Supported Database Specifiers

Specifier	Supported Drivers	Example

Oracle Net connection descriptor	Thin, OCI	Thin, using an address list:
		`url="jdbc:oracle:thin:@(DESCRIPTION=`
		`(LOAD_BALANCE=on)`
		`(ADDRESS_LIST=`
		`(ADDRESS=(PROTOCOL=TCP)(HOST=host1) (PORT=5221))`
		`(ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=5221)))`
		`(CONNECT_DATA=(SERVICE_NAME=orcl)))"`
		OCI, using a cluster:
		`"jdbc:oracle:oci:@(DESCRIPTION=`
		`(ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias)`
		`(PORT=5221))`
		`(CONNECT_DATA=(SERVICE_NAME=orcl)))"`

Thin-style service name	Thin	Refer to "Thin-style Service Name Syntax" for details. `"jdbc:oracle:thin:HR/hr@//localhost:5221/orcl"`
LDAP syntax	Thin	Refer to LDAP Syntax for details. `"jdbc:oracle:thin:@ldap://ldap.example.com:7777/sales,cn=OracleContext,dc=com"`
Bequeath connection	OCI	Empty. That is, nothing after @ `"jdbc:oracle:oci:HR/hr/@"`

TNSNames alias	Thin, OCI	Refer to "TNSNames Alias Syntax" for details.
		`OracleDataSource ods = new OracleDataSource();`
		`ods.setTNSEntryName("MyTNSAlias");`

Thin-style Service Name Syntax

Thin-style service names are supported only by the JDBC Thin driver. The syntax is:

```
@//host_name:port_number/service_name
```

For example:

```
jdbc:oracle:thin:HR/hr@//localhost:5221/orcl
```

Note: The JDBC Thin driver supports only the TCP/IP protocol.

Support for Delay in Connection Retries

Oracle Database 12*c* Release 1 (12.1.0.2) introduces a new connection attribute `RETRY_DELAY`. The `RETRY_DELAY` attribute specifies the delay between connection retries in seconds. The following code snippet shows how to use this attribute:

```
(DESCRIPTION_LIST=
  (DESCRIPTION=
   (CONNECT_TIMEOUT=10)(RETRY_COUNT=3)(RETRY_DELAY=3)
   (ADDRESS_LIST=
    (ADDRESS=(PROTOCOL=tcp)(HOST=myhost1)(PORT=1521))
    (ADDRESS=(PROTOCOL=tcp)(HOST=myhost2)(PORT=1521)))
   (CONNECT_DATA=(SERVICE_NAME=example1.com)))
  (DESCRIPTION=
   (CONNECT_TIMEOUT=60)(RETRY_COUNT=1)(RETRY_DELAY=5)
   (ADDRESS_LIST=
    (ADDRESS=(PROTOCOL=tcp)(HOST=myhost3)(PORT=1521))
    (ADDRESS=(PROTOCOL=tcp)(HOST=myhost4)(PORT=1521)))
   (CONNECT_DATA=(SERVICE_NAME=example2.com))))
```

TNSNames Alias Syntax

You can find the available TNSNAMES entries listed in the tnsnames.ora file on the client computer from which you are connecting. On Windows, this file is located in the *ORACLE_HOME*\NETWORK\ADMIN directory. On UNIX systems, you can find it in the *ORACLE_HOME* directory or the directory indicated in your TNS_ADMIN environment variable.

For example, if you want to connect to the database on host myhost as user HR with password hr that has a TNSNAMES entry of MyHostString, then write the following:

```
OracleDataSource ods = new OracleDataSource();
ods.setTNSEntryName("MyTNSAlias");
ods.setUser("HR");
ods.setPassword("hr");
ods.setDriverType("oci");
Connection conn = ods.getConnection();
```

The oracle.net.tns_admin system property must be set to the location of the tnsnames.ora file so that the JDBC Thin driver can locate the tnsnames.ora file. For example:

```
System.setProperty("oracle.net.tns_admin", "c:\\Temp");
String url = "jdbc:oracle:thin:@tns_entry";
```

Note: When using TNSNames with the JDBC Thin driver, you must set the oracle.net.tns_admin property to the directory that contains your tnsnames.ora file.

```
java -Doracle.net.tns_admin=$ORACLE_HOME/network/admin
```

LDAP Syntax

An example of database specifier using the Lightweight Directory Access Protocol (LDAP) syntax is as follows:

```
"jdbc:oracle:thin:@ldap://ldap.example.com:7777/sales,cn=OracleContext,dc=com"
```

When using SSL, change this to:

```
"jdbc:oracle:thin:@ldaps://ldap.example.com:7777/sales,cn=OracleContext,dc=com"
```

Note: The JDBC Thin driver can use LDAP over SSL to communicate with Oracle Internet Directory if you substitute `ldaps:` for `ldap:` in the database specifier. The LDAP server must be configured to use SSL. If it is not, then the connection attempt will hang.

The JDBC Thin driver supports failover of a list of LDAP servers during the service name resolution process, without the need for a hardware load balancer. Also, client-side load balancing is supported for connecting to LDAP servers. A list of space separated LDAP URLs syntax is used to support failover and load balancing.

When a list of LDAP URLs is specified, both failover and load balancing are enabled by default. The `oracle.net.ldap_loadbalance` connection property can be used to disable load balancing, and the `oracle.net.ldap_failover` connection property can be used to disable failover.

An example, which uses failover, but with client-side load balancing disabled, is as follows:

```
Properties prop = new Properties();
String url =
"jdbc:oracle:thin:@ldap://ldap1.example.com:3500/cn=salesdept,cn=OracleContext,dc=
com/salesdb " +
"ldap://ldap2.example.com:3500/cn=salesdept,cn=OracleContext,dc=com/salesdb " +
"ldap://ldap3.example.com:3500/cn=salesdept,cn=OracleContext,dc=com/salesdb";

prop.put("oracle.net.ldap_loadbalance", "OFF" );
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setConnectionProperties(prop);
```

The JDBC Thin driver supports LDAP nonanonymous bind. A set of JNDI environment properties, which contains authentication information, can be specified for a data source. If an LDAP server is configured as not to allow anonymous bind, then authentication information must be provided to connect to the LDAP server. The following example shows a simple clear-text password authentication:

```
String url =
"jdbc:oracle:thin:@ldap://ldap.example.com:7777/sales,cn=salesdept,cn=OracleContex
t,dc=com";

Properties prop = new Properties();
prop.put("java.naming.security.authentication", "simple");
prop.put("java.naming.security.principal","cn=salesdept,cn=OracleContext,dc=com");
prop.put("java.naming.security.credentials", "mysecret");

OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setConnectionProperties(prop);
```

Since JDBC passes down the three properties to JNDI, the authentication mechanism chosen by client is consistent with how these properties are interpreted by JNDI. For example, if the client specifies authentication information without explicitly specifying the `java.naming.security.authentication` property, then the default authentication mechanism is "simple". Please refer to relevant JDNI documentation for details.

9

JDBC Client-Side Security Features

This chapter discusses support in the Oracle Java Database Connectivity (JDBC) Oracle Call Interface (OCI) and JDBC Thin drivers for login authentication, data encryption, and data integrity, particularly, with respect to features of the Oracle Advanced Security option.

Oracle Advanced Security, previously known as the Advanced Networking Option (ANO) or Advanced Security Option (ASO), provides industry standards-based data encryption, data integrity, third-party authentication, single sign-on, and access authorization. Starting from Oracle Database 11g Release 1, both the JDBC OCI and JDBC Thin drivers support all the Oracle Advanced Security features.

> **Note:** This discussion is not relevant to the server-side internal driver because all communication through server-side internal driver is completely internal to the server.

This chapter contains the following sections:

Support for Oracle Advanced Security

Oracle Advanced Security provides the following security features:

- Data Encryption

  Sensitive information communicated over enterprise networks and the Internet can be protected by using encryption algorithms, which transform information into a form that can be deciphered only with a decryption key. Some of the supported encryption algorithms are RC4, DES, 3DES, and AES.

To ensure data integrity during transmission, Oracle Advanced Security generates a cryptographically secure message digest. Starting from Oracle Database 12*c* Release 1 (12.1), the SHA-2 list of hashing algorithms are also supported and Oracle Advanced Security uses the following hashing algorithms to generate the secure message digest and includes it with each message sent across a network:

- MD5
- SHA1
- SHA256
- SHA384
- SHA512

This protects the communicated data from attacks, such as data modification, deleted packets, and replay attacks.

The following code snippet shows how to calculate the checksum using any of the algorithms mentioned previously:

```
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES,
"( MD5, SHA1, SHA256, SHA384 or SHA512 )");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,
"REQUIRED");
```

- Strong Authentication

  To ensure network security in distributed environments, it is necessary to authenticate the user and check his credentials. Password authentication is the most common means of authentication. Oracle Database enables strong authentication with Oracle authentication adapters, which support various third-party authentication services, including SSL with digital certificates. Oracle Database supports the following industry-standard authentication methods:

  - Kerberos
  - Remote Authentication Dial-In User Service (RADIUS)
  - Secure Sockets Layer (SSL)

  See Also: *Oracle Database Security Guide*

JDBC OCI Driver Support for Oracle Advanced Security

If you are using the JDBC OCI driver, which presumes that you are running from a computer with an Oracle client installation, then support for Oracle Advanced Security and incorporated third-party features is fairly similar to the support provided by in any Oracle client situation. Your use of Advanced Security features is determined by related settings in the `sqlnet.ora` file on the client computer.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file `oraaccess.xml` instead of the OCI-specific configuration parameters present in the `sqlnet.ora` file. However, the configuration parameters present in the `sqlnet.ora` file are still supported. For more information about the `oraaccess.xml` file, see *Oracle Call Interface Programmer's Guide*.

Starting from Oracle Database 11*g* Release 1, the JDBC OCI driver attempts to use external authentication if you try connecting to a database without providing a password. The following are some examples using the JDBC OCI driver to connect to a database without providing a password:

SSL Authentication

Example 9–1 Using SSL authentication to connect to the database.

Example 9–1 Using SSL Authentication to Connect to the Database

```
import java.sql.*;
import java.util.Properties;

public class test
{
    public static void main( String [] args ) throws Exception
    {
        String url = "jdbc:oracle:oci:@"
         +"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=localhost)(PORT=5221))"
         +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";
        Driver driver = new oracle.jdbc.OracleDriver();
        Properties props = new Properties();
        Connection conn = driver.connect( url, props );
        conn.close();
    }
}
```

Using Data Source

Example 9–2 uses a data source to connect to the database.

Example 9–2 Using a Data Source to Connect to the Database

```
import java.sql.*;
import javax.sql.*;
import java.util.Properties;
import oracle.jdbc.pool.*;

public class testpool {
    public static void main( String args ) throws Exception
    { String url = "jdbc:oracle:oci:@"
+"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=localhost)(PORT=5221))"
 +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";
    OracleConnectionPoolDataSource ocpds = new OracleConnectionPoolDataSource();
    ocpds.setURL(url);
    PooledConnection pc = ocpds.getPooledConnection();
    Connection conn = pc.getConnection();
    }
    }
```

JDBC Thin Driver Support for Oracle Advanced Security

The JDBC Thin driver cannot assume the existence of an Oracle client installation or the presence of the `sqlnet.ora` file. Therefore, it uses a Java approach to support Oracle Advanced Security. Java classes that implement Oracle Advanced Security are included in the `ojdbc6.jar` and `ojdbc7.jar` files. Security parameters for encryption and integrity, usually set in the `sqlnet.ora` file, are set using a Java `Properties` object or through system properties.

Support for Login Authentication

Basic login authentication through JDBC consists of user names and passwords, as with any other means of logging in to an Oracle server. Specify the user name and password through a Java properties object or directly through the `getConnection` method call. This applies regardless of which client-side Oracle JDBC driver you are using, but is irrelevant if you are using the server-side internal driver, which uses a special direct connection and does not require a user name or password.

Starting with Oracle Database 12*c* Release 1 (12.1.0.2), the Oracle JDBC Thin driver supports the `O7L_MR` client ability when you are running your application with a JDK such as JDK 8, which supports the `PBKDF2-SHA2` algorithm. If you are running an application with JDK 7, then you must add a third-party security provider that supports the `PBKDF2-SHA2` algorithm, otherwise the driver will not support the new `12a` password verifier that requires the `O7L_MR` client ability.

If you are using Oracle Database 12*c* Release 1 (12.1.0.2) with the `SQLNET.ALLOWED_LOGON_VERSION_SERVER` parameter set to `12a`, then keep the following points in mind:

- You must also use the 12.1.0.2 Oracle JDBC Thin driver and JDK 8 or JDK 7 with a third-party security provider that supports the `PBKDF2-SHA2` algorithm
- If you use an earlier version of Oracle JDBC Thin driver, then you will get the following error:

  ```
  ORA-28040: No matching authentication protocol
  ```

- If you use the 12.1.0.2 Oracle JDBC Thin driver with JDK 7, then also you will get the same error, if you do not add a third-party security provider that supports the `PBKDF2-SHA2` algorithm.

Support for Strong Authentication

Oracle Advanced Security enables Oracle Database users to authenticate externally. External authentication can be with RADIUS, Kerberos, Certificate-Based Authentication, Token Cards, and Smart Cards. This is called strong authentication. Oracle JDBC drivers provide support for the following strong authentication methods:

- Kerberos
- RADIUS
- SSL (certificate-based authentication)

> **See Also:** *Oracle Database Net Services Reference* for more information about the `SQLNET.ALLOWED_LOGON_VERSION_SERVER` parameter

Support for OS Authentication

Operating System (OS) authentication feature enables Oracle server to pass control of user authentication to the operating system. Using this feature, you can connect to the database by authenticating the OS user name in the database. There is no password for the account associated with the OS user name because it is assumed that OS authentication is sufficient. In this case, the server delegates the authentication to the client OS. You must perform the following steps to use this feature:

- Use the following command to check the value of the Oracle `OS_AUTHENT_PREFIX` initialization parameter:

  ```
  SQL> SHOW PARAMETER os_authent_prefix
  ```

```
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
os_authent_prefix                    string      ops$
SQL>
```

> **Note:** Remember the OS authentication prefix because you must create a database user to enable an OS authenticated connection, where the user name must be the prefix value concatenated to the OS user name.

- Add the following line in the `t_init1.ora` file:

```
REMOTE_OS_AUTHENT = TRUE
```

When a connection is attempted from the local database server, then the OS user name is passed to the Oracle server. If the user name is recognized, then the connection is accepted, otherwise the connection is rejected.

Configuration Steps for Linux

Perform the following steps to set up OS authentication on Linux:

1. Use the following commands to create an OS user `w_smith`:

```
# useradd w_smith
# passwd w_smith
Changing password for w_smith
New password: password
Retype new password: password
```

2. Use the following command to create a database user who can use an OS authenticated connection:

```
CREATE USER ops$w_smith IDENTIFIED EXTERNALLY;
GRANT CONNECT TO ops$w_smith;
```

3. Use the following commands to test the OS authentication connection:

```
su - w_smith
export ORACLE_HOME=/u01/app/oracle/product/12.1.0/db_1
export PATH=$PATH:$ORACLE_HOME/bin
export ORACLE_SID=orcl
sqlplus /
```

Configuration Steps for Windows

Perform the following steps to set up OS authentication on Windows:

> **Note:** Oracle JDBC Thin drivers do not support NTS.

1. Use the following steps to create a local user, say, `w_smith`, using the Computer Management window:
   1. Click **Start.**

2. From the Start menu, select **Programs,** then select **Administrative Tools** and then select **Computer Management.**
3. Clicking on the preceding Plus ("+") sign to expand **Local Users and Groups** on the left pane.
4. Click **Users.**
5. Select **New User** from the Action menu.
6. Enter details of the user in the New User dialog box and click **Create.**

Note: The preceding steps are only for creating a local user. Domain users can be created in Active Directory.

---

2. Use the following command to create a database user who can use an OS authenticated connection:

```
CREATE USER "OPS$yourdomain.com\w_smith" IDENTIFIED EXTERNALLY;
GRANT CONNECT TO "OPS$yourdomain.com\w_smith";
```

Note: When you create the database user in Windows environment, the user name should be in the following format:

```
<OS_authentication_prefix_parameter>$<DOMAIN>\<OS_user_name>
```

When using a Windows server, there is an additional consideration. The following option must be set in the `%ORACLE_HOME%\network\admin\sqlnet.ora` file:

```
sqlnet.authentication_services= (nts)
```

3. Use the following commands to test the OS authentication connection:

```
C:\> set ORACLE_SID=orcl
C:\> sqlplus /
```

JDBC Code Using OS Authentication

Now that you have set up OS authentication to connect to the database, you can use the following JDBC code for connecting to the database:

```
String url = "jdbc:oracle:thin:@oracleserver.mydomain.com:5521:orcl"
Properties props = new Properties();
Connection conn = DriverManager.getConnection( url, props);
```

The preceding code assumes that it is executed by `w_smith` on the client machine. The JDBC drivers retrieve the OS user name from the `user.name` system property that is set by the JVM. As a result, the following thin driver-specific error no longer exists:

```
ORA-17443=Null user or password not supported in THIN driver
```

Note: By default, the JDBC driver retrieves the OS user name from the `user.name` system property, which is set by the JVM. If the JDBC driver is unable to retrieve this system property or if you want to override the value of this system property, then you can use the `OracleConnection.CONNECTION_PROPERTY_THIN_VSESSION_OSUSER` connection property. For more information, refer to *Oracle Database JDBC Java API Reference*.

Support for Data Encryption and Integrity

You can use Oracle Database and Oracle Advanced Security data encryption and integrity features in your Java database applications, depending on related settings in the server. When using the JDBC OCI driver, set parameters as you would in any Oracle client situation. When using the Thin driver, set parameters through a Java properties object.

Encryption is enabled or disabled based on a combination of the client-side encryption-level setting and the server-side encryption-level setting. Similarly, integrity is enabled or disabled based on a combination of the client-side integrity-level setting and the server-side integrity-level setting.

Encryption and integrity support the same setting levels, `REJECTED`, `ACCEPTED`, `REQUESTED`, and `REQUIRED`. Table 9–1 shows how these possible settings on the client-side and server-side combine to either enable or disable the feature. By default, remote OS authentication (through TCP) is disabled in the database for security reasons.

Table 9–1 Client/Server Negotiations for Encryption or Integrity

	Client Rejected	Client Accepted (default)	Client Requested	Client Required
Server Rejected	OFF	OFF	OFF	connection fails
Server Accepted (default)	OFF	OFF	ON	ON
Server Requested	OFF	ON	ON	ON
Server Required	connection fails	ON	ON	ON

Table 9–1 shows, for example, that if encryption is requested by the client, but rejected by the server, it is disabled. The same is true for integrity. As another example, if encryption is accepted by the client and requested by the server, it is enabled. The same is also true for integrity.

See Also:

- *Oracle Database Advanced Security Guide* for more information about Transparent Data Encryption (TDE)
- *Oracle Database Security Guide* for more information about data encryption and integrity features, except TDE

Note: The term checksum still appears in integrity parameter names, but is no longer used otherwise. For all intents and purposes, checksum and integrity are synonymous.

This section covers the following topics:

- JDBC OCI Driver Support for Encryption and Integrity
- JDBC Thin Driver Support for Encryption and Integrity
- Setting Encryption and Integrity Parameters in Java

JDBC OCI Driver Support for Encryption and Integrity

If you are using the JDBC OCI driver, which presumes an Oracle-client setting with an Oracle client installation, then you can enable or disable data encryption or integrity and set related parameters as you would in any Oracle client situation, through settings in the `sqlnet.ora` file on the client.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file `oraaccess.xml` instead of the OCI-specific configuration parameters present in the `sqlnet.ora` file. However, the configuration parameters present in the `sqlnet.ora` file are still supported. For more information about the `oraaccess.xml` file, see *Oracle Call Interface Programmer's Guide*.

To summarize, the client parameters are shown in Table 9–2:

Table 9–2 OCI Driver Client Parameters for Encryption and Integrity

Parameter Description	Parameter Name	Possible Settings

Client encryption level	`SQLNET.ENCRYPTION_CLIENT`	`REJECTED`
		`ACCEPTED`
		`REQUESTED`
		`REQUIRED`

Client encryption selected list	`SQLNET.ENCRYPTION_TYPES_CLIENT`	`RC4_40`, `RC4_56`, `DES`, `DES40`, `AES128`, `AES192`, `AES256`, `3DES112`, `3DES168` (see Note)

Client integrity level	`SQLNET.CRYPTO_CHECKSUM_CLIENT`	`REJECTED`
		`ACCEPTED`
		`REQUESTED`
		`REQUIRED`

Client integrity selected list	`SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT`	`MD5`, `SHA-1`

Note: For the Oracle Advanced Security domestic edition only, settings of `RC4_128` and `RC4_256` are also possible.

See Also: *Oracle Database Security Guide* for more information about configuring network data encryption and integrity

JDBC Thin Driver Support for Encryption and Integrity

The JDBC Thin driver support for data encryption and integrity parameter settings parallels the JDBC OCI driver support discussed in the preceding section. You can set the corresponding parameters through a Java properties object that you can use while opening a database connection.

The default value for the encryption and integrity level is `ACCEPTED` for both the server side and the client side. This enables you to achieve the desired security level for a connection pair by configuring only one side of a connection, either the server side or the client side. This increases the efficiency of your program because if there are multiple Oracle clients connecting to an Oracle Server, then you need to change the encryption and integrity level to `REQUESTED` in the `sqlnet.ora` file only on the server side to turn on encryption or integrity for all connections. This saves time and effort because you do not have to change the settings for each client separately.

Table 9–3 lists the parameter information for the JDBC Thin driver. These parameters are defined in the `oracle.jdbc.OracleConnection` interface.

Table 9–3 Thin Driver Client Parameters for Encryption and Integrity

Parameter Name	Parameter Type	Possible Settings

`CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL`	`String`	`REJECTED`
		`ACCEPTED`
		`REQUESTED`
		`REQUIRED`

`CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES`	`String`	`AES256`, `AES192`, `AES128`, `3DES168`, `3DES112`, `DES56C`, `DES40C`, `RC4_256`, `RC4_128`, `RC4_40`, `RC4_56`

`CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL`	`String`	`REJECTED`
		`ACCEPTED`
		`REQUESTED`
		`REQUIRED`

`CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES`	`String`	`MD5`, `SHA1`, SHA256, SHA384, SHA512

Note:

- Oracle Advanced Security support for the Thin driver is incorporated directly into the JDBC classes JAR file. So, there is no separate version for domestic and export editions. Only parameter settings that are suitable for an export edition are possible.
- The letter C in `DES40C` and `DES56C` refers to Cipher Block Chaining (CBC) mode.

Setting Encryption and Integrity Parameters in Java

Use a Java properties object, that is, an instance of `java.util.Properties`, to set the data encryption and integrity parameters supported by the JDBC Thin driver.

The following example instantiates a Java properties object, uses it to set each of the parameters in Table 9–3, and then uses the properties object in opening a connection to the database:

```
...
Properties prop = new Properties();
```

```
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL,
"REQUIRED");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES,
"( DES40C )");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,
"REQUESTED");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES, "(
MD5 )");

OracleDataSource ods = new OracleDataSource();
ods.setProperties(prop);
ods.setURL("jdbc:oracle:thin:@localhost:5221:main");
Connection conn = ods.getConnection();
...
```

The parentheses around the values encryption type and checksum type allow for lists of values. When multiple values are supplied, the server and the client negotiate to determine which value is to be actually used.

Example

Example 9–3 is a complete class that sets data encryption and integrity parameters before connecting to a database to perform a query.

> **Note:** In the example, the string `REQUIRED` is retrieved dynamically through functionality of the `AnoServices` and `Service` classes. You have the option of retrieving the strings in this manner or including them in the software code as shown in the previous examples

Before running this example, you must turn on encryption in the `sqlnet.ora` file. For example, the following lines will turn on AES256, AES192, and AES128 for the encryption and MD5 and SHA1 for the checksum:

```
SQLNET.ENCRYPTION_SERVER = ACCEPTED
SQLNET.CRYPTO_CHECKSUM_SERVER = ACCEPTED
SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER = (MD5, SHA1)
SQLNET.ENCRYPTION_TYPES_SERVER = (AES256, AES192, AES128)
SQLNET.CRYPTO_SEED = 2z0hslkdharUJCFtkwbjOLbgwsj7vkqt3bGoUylihnvkhgkdsbdskkKGhdk
```

Example 9–3 Setting Data Encryption and Integrity Parameters

```
import java.sql.*;
import java.util.Properties;
import oracle.net.ano.AnoServices;
import oracle.jdbc.*;

public class DemoAESAndSHA1
{
  static final String USERNAME= "HR";
  static final String PASSWORD= "hr";
  static final String URL =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=localhost)(PORT=5221)
)"
  +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";

  public static final void main(String[] argv)
  {
    DemoAESAndSHA1 demo = new DemoAESAndSHA1();
```

```
    try
    {
      demo.run();
    }catch(SQLException ex)
    {
      ex.printStackTrace();
    }
  }

  void run() throws SQLException
  {
    OracleDriver dr = new OracleDriver();
    Properties prop = new Properties();

    // We require the connection to be encrypted with either AES256 or AES192.
    // If the database doesn't accept such a security level, then the connection
attempt will fail.

    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_
LEVEL,AnoServices.ANO_REQUIRED);
    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_
TYPES,"( " + AnoServices.ENCRYPTION_AES256
     + "," + AnoServices.ENCRYPTION_AES192 + ")");

    // We also require the use of the SHA1 algorithm for data integrity checking.

    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_
LEVEL,AnoServices.ANO_REQUIRED);
    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES,
"( " + AnoServices.CHECKSUM_SHA1 + " )");
    prop.setProperty("user",DemoAESAndSHA1.USERNAME);
    prop.setProperty("password",DemoAESAndSHA1.PASSWORD);
    OracleConnection oraConn =
(OracleConnection)dr.connect(DemoAESAndSHA1.URL,prop);
    System.out.println("Connection created! Encryption algorithm is: " +
oraConn.getEncryptionAlgorithmName() + ", data
    integrity algorithm is: " + oraConn.getDataIntegrityAlgorithmName());
    oraConn.close();
  }

}
```

Support for SSL

Oracle Database 12*c* provides support for the Secure Sockets Layer (SSL) protocol. SSL is a widely used industry standard protocol that provides secure communication over a network. SSL provides authentication, data encryption, and data integrity. It provides a secure enhancement to the standard TCP/IP protocol, which is used for Internet communication.

SSL uses digital certificates that comply with the X.509v3 standard for authentication and a public and private key pair for encryption. SSL also uses secret key cryptography and digital signatures to ensure privacy and integrity of data. When a network connection over SSL is initiated, the client and server perform an SSL handshake that includes the following steps:

- Client and server negotiate about the cipher suites to use. This includes deciding on the encryption algorithms to be used for data transfer.

- Server sends its certificate to the client, and the client verifies that the certificate was signed by a trusted certification authority (CA). This step verifies the identity of the server.
- If client authentication is required, the client sends its own certificate to the server, and the server verifies that the certificate was signed by a trusted CA.
- Client and server exchange key information using public key cryptography. Based on this information, each generates a session key. All subsequent communications between the client and the server is encrypted and decrypted by using this set of session keys and the negotiated cipher suite.

Note: In Oracle Database 11*g* Release 1 (11.1), SSL authentication is supported in the thin driver. So, you do not need to provide a user name/password pair if you are using SSL authentication.

SSL Terminology

The following terms are commonly used in the SSL context:

- **certificate**: A certificate is a digitally signed document that binds a public key with an entity. The certificate can be used to verify that the public key belongs to that individual.
- **certification authority**: A certification authority (CA), also known as certificate authority, is an entity which issues digitally signed certificates for use by other parties.
- **cipher suite**: A cipher suite is a set of cryptographic algorithms and key sizes used to encrypt data sent over an SSL-enabled network.
- **private key**: A private key is a secret key, which is never transmitted over a network. The private key is used to decrypt a message that has been encrypted using the corresponding public key. It is also used to sign certificates. The certificate is verified using the corresponding public key.
- **public key**: A public key is an encryption key that can be made public or sent by ordinary means such as an e-mail message. The public key is used for encrypting the message sent over SSL. It is also used to verify a certificate signed by the corresponding private key.
- **wallet**: A wallet is a password-protected container that is used to store authentication and signing credentials, including private keys, certificates, and trusted certificates required by SSL.

Java Version of SSL

The Java Secure Socket Extension (JSSE) provides a framework and an implementation for a Java version of the SSL and TLS protocols. JSSE provides support for data encryption, server and client authentication, and message integrity. It abstracts the complex security algorithms and handshaking mechanisms and simplifies application development by providing a building block for application developers, which they can directly integrate into their applications. JSSE is integrated into Java Development Kit (JDK) 1.4 and later, and supports SSL version 2.0 and 3.0.

Oracle strongly recommends that you have a clear understanding of the JavaTM Secure Socket Extension (JSSE) framework before using SSL in the Oracle JDBC drivers.

The JSSE standard application programming interface (API) is available in the `javax.net`, `javax.net.ssl`, and `javax.security.cert` packages. These packages provide classes for creating and configuring sockets, server sockets, SSL sockets, and SSL server sockets. The packages also provide a class for secure HTTP connections, a public key certificate API compatible with JDK1.1-based platforms, and interfaces for key and trust managers.

SSL works the same way, as in any networking environment, in Oracle Database 12*c* Release 1 (12.1). This section covers the following:

- Managing Certificates and Wallets
- Keys and certificates containers

Managing Certificates and Wallets

To establish an SSL connection with a JDBC client, Thin or OCI, Oracle database server sends its certificate, which is stored in its wallet. The client may or may not need a certificate or wallet depending on the server configuration.

The Oracle JDBC Thin driver uses the JSSE framework to create an SSL connection. It uses the default provider (*SunJSSE*) to create an SSL context. However you can provide your own provider.

You do not need a certificate for the client, unless the `SSL_CLIENT_AUTHENTICATION` parameter is set on the server.

Keys and certificates containers

Java clients can use multiple types of containers such as Oracle wallets, JKS, PKCS12, and so on, as long as a provider is available. For Oracle wallets, *OraclePKI* provider must be used because the PKCS12 support provided by *SunJSSE* provider does not support all the features of PKCS12. In order to use *OraclePKI* provider, the following JARs are required:

- `oraclepki.jar`
- `osdt_cert.jar`
- `osdt_core.jar`

All these JAR files should be under `$ORACLE_HOME/jlib directory`.

Support for Kerberos

Kerberos is a network authentication protocol that provides the tools of authentication and strong cryptography over the network. Kerberos helps you secure your information systems across your entire enterprise by using secret-key cryptography. The Kerberos protocol uses strong cryptography so that a client or a server can prove its identity to its server or client across an insecure network connection. After a client and server have used Kerberos to prove their identity, they can also encrypt all of their communications to assure privacy and data integrity as they go about their business.

The Kerberos architecture is centered around a trusted authentication service called the key distribution center, or KDC. Users and services in a Kerberos environment are referred to as principals; each principal shares a secret, such as a password, with the KDC. A principal can be a user such as `HR` or a database server instance.

Starting from 12c Release 1, Oracle Database also supports cross-realm authentication for Kerberos. If you add the referred realm appropriately in the `domain_realms` section

of the kerberos configuration file, then being in one particular realm, you can access the services of another realm.

This section contains the following subsections:

- Configuring Windows to Use Kerberos
- Configuring Oracle Database to Use Kerberos

Configuring Windows to Use Kerberos

A good Kerberos client providing `klist, kinit,` and other tools, can be found at the following link:

`http://web.mit.edu/kerberos/dist/index.html`

This client also provides a nice GUI.

You need to make the following changes to configure Kerberos on your Windows machine:

1. Right-click the **My Computer** icon on your desktop.
2. Select **Properties**. The System Properties dialog box is displayed.
3. Select the **Advanced** tab.
4. Click **Environment Variables**. The Environment Variables dialog box is displayed.
5. Click **New** to add a new user variable. The New User Variable dialog box is displayed.
6. Enter `KRB5CCNAME` in the Variable name field.
7. Enter `FILE:C:\Documents and Settings\<user_name>\krb5cc` in the Variable value field.
8. Click **OK** to close the New User Variable dialog box.
9. Click **OK** to close the Environment Variables dialog box.
10. Click **OK** to close the System Properties dialog box.

Note: `C:\WINDOWS\krb5.ini` file has the same content as `krb5.conf` file.

Configuring Oracle Database to Use Kerberos

Perform the following steps to configure Oracle Database to use Kerberos:

1. Use the following command to connect to the database:

```
SQL> connect system
Enter password: password
```

2. Use the following commands to create a user `CLIENT@US.ORACLE.COM` that is identified externally:

```
SQL> create user "CLIENT@US.ORACLE.COM" identified externally;
SQL> grant create session to "CLIENT@US.ORACLE.COM";
```

3. Use the following commands to connect to the database as `sysdba` and dismount it:

```
SQL> connect / as sysdba
```

```
SQL> shutdown immediate;
```

4. Add the following line to `$T_WORK/t_init1.ora` file:

```
OS_AUTHENT_PREFIX=""
```

5. Use the following command to restart the database:

```
SQL> startup pfile=t_init1.ora
```

6. Modify the `sqlnet.ora` file to include the following lines:

```
names.directory_path = (tnsnames)
#Kerberos
sqlnet.authentication_services = (beq,kerberos5)
sqlnet.authentication_kerberos5_service = dbji
sqlnet.kerberos5_conf = /home/Jdbc/Security/kerberos/krb5.conf
sqlnet.kerberos5_keytab = /home/Jdbc/Security/kerberos/dbji.oracleserver
sqlnet.kerberos5_conf_mit = true
sqlnet.kerberos_cc_name = /tmp/krb5cc_5088
# logging (optional):
trace_level_server=16
trace_directory_server=/scratch/sqlnet/
```

7. Use the following commands to verify that you can connect through SQL*Plus:

```
> kinit client
> klist
     Ticket cache: FILE:/tmp/krb5cc_5088
     Default principal: client@US.ORACLE.COM

     Valid starting     Expires            Service principal
     06/22/06 07:13:29  06/22/06 17:13:29  krbtgt/US.ORACLE.COM@US.ORACLE.COM


     Kerberos 4 ticket cache: /tmp/tkt5088
     klist: You have no tickets cached
> sqlplus
'/@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oracleserver.mydomain.com)(PORT=52
21))
(CONNECT_DATA=(SERVICE_NAME=orcl)))'
```

Note: For information about using Kerberos, refer to the following web sites

http://technet.microsoft.com/en-us/windowsserver/bb512919
http://access.redhat.com/knowledge/docs/

Code Example

This following example demonstrates the new Kerberos authentication feature that is part of Oracle Database 12*c* Release 1 (12.1) JDBC thin driver. This demo covers two scenarios:

- In the first scenario, the OS maintains the user name and credentials. The credentials are stored in the cache and the driver retrieves the credentials before trying to authenticate to the server. This scenario is in the module `connectWithDefaultUser()`.

Note:

1. Before you run this part of the demo, use the following command to verify that you have valid credentials:

   ```
   > /usr/kerberos/bin/kinit client
   where, the password is welcome.
   ```

2. Use the following command to list your tickets:

   ```
   > /usr/kerberos/bin/klist
   ```

- The second scenario covers the case where the application wants to control the user credentials. This is the case of the application server where multiple web users have their own credentials. This scenario is in the module `connectWithSpecificUser()`.

Note: To run this demo, you need to have a working setup, that is, a Kerberos server up and running, and an Oracle database server that is configured to use Kerberos authentication. You then need to change the URLs used in the example to compile and run it.

Example 9–4 Using Kerberos Authentication to Connect to the Database

```
import com.sun.security.auth.module.Krb5LoginModule;
import java.io.IOException;

import java.security.PrivilegedExceptionAction;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import java.util.HashMap;
import java.util.Properties;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;

import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.net.ano.AnoServices;
public class KerberosJdbcDemo
{
  String url ="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)"+
    "(HOST=oracleserver.mydomain.com)(PORT=5221))(CONNECT_DATA=" +
    "(SERVICE_NAME=orcl)))";

  public static void main(String[] arv)
  {
    /* If you see the following error message [Mechanism level: Could not load
     * configuration file c:\winnt\krb5.ini (The system cannot find the path
     * specified] it's because the JVM cannot locate your kerberos config file.
     * You have to provide the location of the file. For example, on Windows,
     * the MIT Kerberos client uses the config file: C\WINDOWS\krb5.ini:
     */
```

```
    // System.setProperty("java.security.krb5.conf","C:\\WINDOWS\\krb5.ini");

System.setProperty("java.security.krb5.conf","/home/Jdbc/Security/kerberos/krb5.co
nf");

    KerberosJdbcDemo kerberosDemo = new KerberosJdbcDemo();
    try
    {
      System.out.println("Attempt to connect with the default user:");
      kerberosDemo.connectWithDefaultUser();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
    try
    {
      System.out.println("Attempt to connect with a specific user:");
      kerberosDemo.connectWithSpecificUser();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }


  void connectWithDefaultUser() throws SQLException
  {
    OracleDriver driver = new OracleDriver();
    Properties prop = new Properties();

    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_
SERVICES,
      "("+AnoServices.AUTHENTICATION_KERBEROS5+")");
    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_
KRB5_MUTUAL,
      "true");

    /* If you get the following error [Unable to obtain Princpal Name for
     * authentication] although you know that you have the right TGT in your
     * credential cache, then it's probably because the JVM can't locate your
     * cache.
     *
     * Note that the default location on windows is "C:\Documents and
Settings\krb5cc_username".
     */

    // prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_
AUTHENTICATION_KRB5_CC_NAME,
    /*
      On linux:
         > which kinit
         /usr/kerberos/bin/kinit
         > ls -l /etc/krb5.conf
         lrwxrwxrwx    1 root  root   47 Jun 22 06:56 /etc/krb5.conf ->
/home/Jdbc/Security/kerberos/krb5.conf

         > kinit client
         Password for client@US.ORACLE.COM:
```

```
        > klist
        Ticket cache: FILE:/tmp/krb5cc_5088
        Default principal: client@US.ORACLE.COM

        Valid starting     Expires            Service principal
        11/02/06 09:25:11  11/02/06 19:25:11  krbtgt/US.ORACLE.COM@US.ORACLE.COM


        Kerberos 4 ticket cache: /tmp/tkt5088
        klist: You have no tickets cached
    */
    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_
KRB5_CC_NAME,
                     "/tmp/krb5cc_5088");
    Connection conn  = driver.connect(url,prop);
    String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
    printUserName(conn);
    conn.close();
  }


  void connectWithSpecificUser() throws Exception
  {
    Subject specificSubject = new Subject();

    // This first part isn't really meaningful to the sake of this demo. In
    // a real world scenario, you have a valid "specificSubject" Subject that
    // represents a web user that has valid Kerberos credentials.
    Krb5LoginModule krb5Module = new Krb5LoginModule();
    HashMap sharedState = new HashMap();
    HashMap options = new HashMap();
    options.put("doNotPrompt","false");
    options.put("useTicketCache","false");
    options.put("principal","client@US.ORACLE.COM");


krb5Module.initialize(specificSubject,newKrbCallbackHandler(),sharedState,options)
;
    boolean retLogin = krb5Module.login();
    krb5Module.commit();
    if(!retLogin)
      throw new Exception("Kerberos5 adaptor couldn't retrieve credentials (TGT)
from the cache");

    // to use the TGT from the cache:
    // options.put("useTicketCache","true");
    // options.put("doNotPrompt","true");
    // options.put("ticketCache","C:\\Documents and Settings\\user\\krb5cc");
    // krb5Module.initialize(specificSubject,null,sharedState,options);


    // Now we have a valid Subject with Kerberos credentials. The second scenario
    // really starts here:
    // execute driver.connect(...) on behalf of the Subject 'specificSubject':
    Connection conn =
      (Connection)Subject.doAs(specificSubject, new PrivilegedExceptionAction()
        {
          public Object run()
          {
```

```
            Connection con = null;
            Properties prop = new Properties();
            prop.setProperty(AnoServices.AUTHENTICATION_PROPERTY_SERVICES,
                             "(" + AnoServices.AUTHENTICATION_KERBEROS5 + ")");
            try
            {
              OracleDriver driver = new OracleDriver();
              con = driver.connect(url, prop);

            } catch (Exception except)
            {
              except.printStackTrace();
            }
            return con;
          }
        });

    String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
    printUserName(conn);
    conn.close();
  }

  void printUserName(Connection conn) throws SQLException
  {
    Statement stmt = null;
    try
    {
      stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("select user from dual");
      while(rs.next())
        System.out.println("User is:"+rs.getString(1));
      rs.close();
    }
    finally
    {
      if(stmt != null)
        stmt.close();
    }
  }
}

class KrbCallbackHandler implements CallbackHandler
{
 public void handle(Callback[] callbacks) throws IOException,
                                                 UnsupportedCallbackException
 {
   for (int i = 0; i < callbacks.length; i++)
   {
     if (callbacks[i] instanceof PasswordCallback)
     {
       PasswordCallback pc = (PasswordCallback)callbacks[i];
       System.out.println("set password to 'welcome'");
       pc.setPassword((new String("welcome")).toCharArray());
     } else
     {
       throw new UnsupportedCallbackException(callbacks[i],
                                              "Unrecognized Callback");
     }
   }
```

```
 }
}
```

Support for RADIUS

Oracle Database 11*g* Release 1 introduced support for Remote Authentication Dial-In User Service (RADIUS). RADIUS is a client/server security protocol that is most widely known for enabling remote authentication and access. Oracle Advanced Security uses this standard in a client/server network environment to enable use of any authentication method that supports the RADIUS protocol. RADIUS can be used with a variety of authentication mechanisms, including token cards and smart cards.

Configuring Oracle Database to Use RADIUS

Perform the following steps to configure Oracle Database to use RADIUS:

1. Use the following command to connect to the database:

   ```
   SQL> connect system
   Enter password: password
   ```

2. Use the following commands to create a new user `aso` from within a database:

   ```
   SQL> create user aso identified externally;
   SQL> grant create session to aso;
   ```

3. Use the following commands to connect to the database as `sysdba` and dismount it:

   ```
   SQL> connect / as sysdba
   SQL> shutdown immediate;
   ```

4. Add the following lines to the `t_init1.ora` file:

   ```
   os_authent_prefix = ""
   ```

   Note: Once the test is over, you need to revert the preceding changes made to the t_init1.ora file.

5. Use the following command to restart the database:

   ```
   SQL> startup pfile=?/work/t_init1.ora
   ```

6. Modify the `sqlnet.ora` file so that it contains only these lines:

   ```
   sqlnet.authentication_services = ( beq, radius)
   sqlnet.radius_authentication = <RADUIUS_SERVER_HOST_NAME>
   sqlnet.radius_authentication_port = 1812
   sqlnet.radius_authentication_timeout = 120
   sqlnet.radius_secret=/home/Jdbc/Security/radius/radius_key
   # logging (optional):
   trace_level_server=16
   trace_directory_server=/scratch/sqlnet/
   ```

7. Use the following command to verify that you can connect through SQL*Plus:

```
>sqlplus
'aso/1234@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oracleserver.mydomain.com)(
PORT=5221))
(CONNECT_DATA=(SERVICE_NAME=orcl)))'
```

Code Example

This example demonstrates the new RADIUS authentication feature that is a part of Oracle Database 12*c* Release 1 (12.1) JDBC thin driver. You need to have a working setup, that is, a RADIUS server up and running, and an Oracle database server that is configured to use RADIUS authentication. You then need to change the URLs given in the example to compile and run it.

Example 9–5 Using RADIUS Authentication to Connect to the Database

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.net.ano.AnoServices;
public class RadiusJdbcDemo
{
  String url ="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)"+
    "(HOST=oracleserver.mydomain.com)(PORT=5221))(CONNECT_DATA=" +
    "(SERVICE_NAME=orcl)))";

  public static void main(String[] arv)
  {
    RadiusJdbcDemo radiusDemo = new RadiusJdbcDemo();
    try
    {
      radiusDemo.connect();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }

  /*
   * This method attempts to logon to the database using the RADIUS
   * authentication protocol.
   *
   * It should print the following output to stdout:
   * -----------------------------------------------------
   * Authentication adaptor=RADIUS
   * User is:ASO
   * -----------------------------------------------------
   */
  void connect() throws SQLException
  {
    OracleDriver driver = new OracleDriver();
    Properties prop = new Properties();
```

```
    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_
SERVICES,
      "("+AnoServices.AUTHENTICATION_RADIUS+")");
    // The user "aso" needs to be properly setup on the radius server with
    // password "1234".
    prop.setProperty("user","aso");
    prop.setProperty("password","1234");

    Connection conn  = driver.connect(url,prop);
    String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
    printUserName(conn);
    conn.close();
  }


  void printUserName(Connection conn) throws SQLException
  {
    Statement stmt = null;
    try
    {
      stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("select user from dual");
      while(rs.next())
        System.out.println("User is:"+rs.getString(1));
      rs.close();
    }
    finally
    {
      if(stmt != null)
        stmt.close();
    }
  }
}
```

Secure External Password Store

As an alternative for large-scale deployments where applications use password credentials to connect to databases, it is possible to store such credentials in a client-side Oracle wallet. An Oracle wallet is a secure software container that is used to store authentication and signing credentials.

Storing database password credentials in a client-side Oracle wallet eliminates the need to embed user names and passwords in application code, batch jobs, or scripts. This reduces the risk of exposing passwords in the scripts and application code, and simplifies maintenance because you do not need to change your code each time user names and passwords change. In addition, if you do not have to change the application code, then it also becomes easier to enforce password management policies for these user accounts.

You can set the `oracle.net.wallet_location` connection property to specify the wallet location. The JDBC driver can then retrieve the user name and password pair from this wallet.

See Also:

- *Oracle Database Administrator's Guide* for more information about configuring your client to use secure external password store and for information about managing credentials in it
- *Oracle Database Security Guide* for more information about managing the secure external password store for password credentials

10
Proxy Authentication

Oracle Java Database Connectivity (JDBC) provides proxy authentication, also called N-tier authentication. This feature is supported through both the JDBC Oracle Call Interface (OCI) driver and the JDBC Thin driver. This chapter contains the following sections:

> **Note:** Oracle Database supports proxy authentication functionality in three tiers *only*. It does not support it across multiple middle tiers.

About Proxy Authentication

Proxy authentication is the process of using a middle tier for user authentication. You can design a middle tier server to proxy clients in a secure fashion by using the following three forms of proxy authentication:

- The middle tier server authenticates itself with the database server and a client. In this case, an application user or another application, authenticates itself with the middle tier server. Client identities can be maintained all the way through to the database.
- The client, that is, a database user, is not authenticated by the middle tier server. The client's identity and database password are passed through the middle tier server to the database server for authentication.
- The client, that is, a global user, is authenticated by the middle tier server, and passes either a Distinguished name (DN) or a Certificate through the middle tier for retrieving the client's user name.

> **Note:** Operations done on behalf of a client by a middle tier server can be audited. For more information, refer to *Oracle Database Security Guide*.

See Also: "Creating Proxy Connections" on page 10-3

In all cases, an administrator must authorize the middle tier server to proxy a client, that is, to act on behalf of the client. Suppose, the middle tier server initially connects to the database as user `HR` and activates a proxy connection as user `jeff`, and then issues the following statement to authorize the middle tier server to proxy a client:

```
ALTER USER jeff GRANT CONNECT THROUGH HR;
```

You can also:

- Specify roles that the middle tier is permitted to activate when connecting as the client. For example,

  ```
  CREATE ROLE role1;
  GRANT SELECT ON employees TO role1;
  ALTER USER jeff GRANT CONNECT THROUGH HR ROLE role1;
  ```

  The role clause limits the access only to those database objects that are mentioned in the list of the roles. The list of roles can be empty.
- Find the users who are currently authorized to connect through a middle tier by querying the `PROXY_USERS` data dictionary view.
- Disallow a proxy connection by using the `REVOKE CONNECT THROUGH` clause of `ALTER USER` statement.

Note: In case of proxy authentication, a JDBC connection to the database creates a database session during authentication, and then other sessions can be created during the life time of the connection.

You need to use the different fields and methods present in the `oracle.jdbc.OracleConnection` interface to set up the different types of proxy connections.

Types of Proxy Connections

You can create proxy connections using any one of the following options:

- `USER NAME`

  This is done by supplying the user name or the password or both. The SQL statement for specifying authentication using password is:

  ```
  ALTER USER jeff GRANT CONNECT THROUGH HR AUTHENTICATED USING password;
  ```

  In this case, `jeff` is the user name and `HR` is the proxy for `jeff`.

  The password option exists for additional security. Having no `authenticated` clause implies default authentication, which is using only the user name without the password. The SQL statement for specifying default authentication is:

  ```
  ALTER USER jeff GRANT CONNECT THROUGH HR
  ```

- `DISTINGUISHED NAME`

  This is a global name in lieu of the password of the user being proxied for. An example of the corresponding SQL statement using a distinguished name is:

  ```
  CREATE USER jeff IDENTIFIED GLOBALLY AS
  ```

```
'CN=jeff,OU=americas,O=oracle,L=redwoodshores,ST=ca,C=us';
```

The string that follows the `identified globally as` clause is the distinguished name. It is then necessary to authenticate using this distinguished name. The corresponding SQL statement to specify authentication using distinguished name is:

```
ALTER USER jeff GRANT CONNECT THROUGH HR AUTHENTICATED USING DISTINGUISHED
NAME;
```

- `CERTIFICATE`

  This is a more encrypted way of passing the credentials of the user, who is to be proxied, to the database. The certificate contains the distinguished name encoded in it. One way of generating the certificate is by creating a wallet and then decoding the wallet to get the certificate. The wallet can be created using `runutl mkwallet`. It is then necessary to authenticate using the generated certificate. The SQL statement for specifying authentication using certificate is:

  ```
  ALTER USER jeff GRANT CONNECT THROUGH HR AUTHENTICATED USING CERTIFICATE;
  ```

Note: The use of certificates for proxy authentication will be desupported in future Oracle Database releases.

Note:

- All the options can be associated with roles.
- When opening a new proxied connection, a new session is started on the Database server. If you start a global transaction and then call the `openProxySession` method, then, at this point, you are no longer a part of the global transaction and instead it is like you are in a freshly created JDBC connection. Typically, this never happens because the `openProxySession` method is called prior to creating or resuming a global transaction. In such a case, you are still a part of the global transaction.

Creating Proxy Connections

A user, say `jeff`, has to connect to the database through another user, say `HR`. The proxy user, `HR`, should have an active authenticated connection. A proxy session is then created on this active connection, with the driver issuing a command to the server to create a session for the user, `jeff`. The server returns the new session ID, and the driver sends a session switch command to switch to this new session.

The JDBC OCI and Thin driver switch sessions in the same manner. The drivers permanently switch to the new session, `jeff`. As a result, the proxy session, `HR`, is not available until the new session, `jeff`, is closed.

Note: You can use the `isProxySession` method from the `oracle.jdbc.OracleConnection` interface to check if the current session associated with your connection is a proxy session. This method returns `true` if the current session associated with the connection is a proxy session.

A new proxy session is opened by using the following method from the `oracle.jdbc.OracleConnection` interface:

```
void openProxySession(int type, java.util.Properties prop) throws
SQLExceptionOpens
```

Where,

`type` is the type of the proxy session and can have the following values:

- `OracleConnection.PROXYTYPE_USER_NAME`

  This type is used for specifying the user name.

- `OracleConnection.PROXYTYPE_DISTINGUISHED_NAME`

  This type is used for specifying the distinguished name of the user.

- `OracleConnection.PROXYTYPE_CERTIFICATE`

  This type is used for specifying the proxy certificate.

`prop` is the property value of the proxy session and can have the following values:

- `PROXY_USER_NAME`

  This property value should be used with the type `OracleConnection.PROXYTYPE_USER_NAME`. The value should be a `java.lang.String`.

- `PROXY_DISTINGUISHED_NAME`

  This property value should be used with the type `OracleConnection.PROXYTYPE_DISTINGUISHED_NAME`. The value should be a `java.lang.String`.

- `PROXY_CERTIFICATE`

  This property value should be used with the type `OracleConnection.PROXYTYPE_CERTIFICATE`. The value is a `bytep[]` array that contains the certificate.

- `PROXY_ROLES`

  This property value can be used with the following types:

  - `OracleConnection.PROXYTYPE_USER_NAME`
  - `OracleConnection.PROXYTYPE_DISTINGUISHED_NAME`
  - `OracleConnection.PROXYTYPE_CERTIFICATE`

  The value should be a `java.lang.String`.

- `PROXY_SESSION`

  This property value is used with the `close` method to close the proxy session.

  See Also: Closing a Proxy Session on page 10-5

- `PROXY_USER_PASSWORD`

  This property value should be used with the type `OracleConnection.PROXYTYPE_USER_NAME`. The value should be a `java.lang.String`.

The following code snippet shows the use of the `openProxySession` method:

```
java.util.Properties prop = new java.util.Properties();
prop.put(OracleConnection.PROXY_USER_NAME, "jeff");
String[] roles = {"role1", "role2"};
prop.put(OracleConnection.PROXY_ROLES, roles);
conn.openProxySession(OracleConnection.PROXYTYPE_USER_NAME, prop);
```

Closing a Proxy Session

You can close the proxy session opened with the `OracleConnection.openProxySession` method by passing the `OracleConnection.PROXY_SESSION` parameter to the `OracleConnection.close` method in the following way:

```
OracleConnection.close(OracleConnection.PROXY_SESSION);
```

This is similar to closing a proxy session on a non-cached connection. The standard `close` method must be called explicitly to close the connection itself. If the `close` method is called directly, without closing the proxy session, then both the proxy session and the connection are closed. This can be achieved in the following way:

```
OracleConnection.close(OracleConnection.INVALID_CONNECTION);
```

Caching Proxy Connections

Proxy connections, like standard connections, can be cached. Caching proxy connections enhances the performance. To cache a proxy connection, you need to create a connection using one of the `getConnection` methods on a cache enabled `OracleDataSource` object.

A proxy connection may be cached in the connection cache using the connection attributes feature of the connection cache. Connection attributes are name/value pairs that are user-defined and help tag a connection before returning it to the connection cache for reuse. When the tagged connection is retrieved, it can be directly used without having to do a round-trip to create or close a proxy session. Universal Connection Pool supports caching of any user/password authenticated connection. Therefore, any user authenticated proxy connection can be cached and retrieved.

It is recommended that proxy connections should not be closed without applying the connection attributes. If a proxy connection is closed without applying the connection attributes, the connection is returned to the connection cache for reuse, but cannot be retrieved. The connection caching mechanism does not remember or reset session state.

A proxy connection can be removed from the connection cache by closing the connection directly.

See Also: "Closing a Proxy Session" on page 10-5

Limitations of Proxy Connections

Closing a proxy connection automatically closes every SQL Statement created by the proxy connection, during the proxy session or prior to the proxy session. This may cause unexpected consequences on application pooling or statement caching. The following code samples explain this limitation of proxy connections:

Example 1

```
....
public void displayName(String N)  // Any function using the Proxy feature
{
     Properties props = new Properties();
     props.put("PROXY_USER_NAME", proxyUser);
```

```
     c.openProxySession(OracleConnection.PROXYTYPE_USER_NAME, props);
     .......
     c.close(OracleConnection.PROXY_SESSION);
}

public static void main (String args[]) throws SQLException
{
    ............
    PreparedStatement pstmt = conn.prepareStatement("SELECT first_name FROM
EMPLOYEES WHERE employee_id = ?");
    pstmt.setInt(1, 205);
    ResultSet rs = pstmt.executeQuery();
    while (rs.next())
    {
        displayName(rs.getString(1));
         if (rs.isClosed() // The ResultSet is already closed while closing the
connection!
         {
             throw new Exception("Your ResultSet has been prematurely closed!
Your Statement object is also dead now.");
         }
    }
}
```

In the preceding example, when you close the proxy connection in the `displayName` method, then the `PreparedStatement` object and the `ResultSet` object also get closed. So, if you do not check the status of the `ResultSet` object inside loop, then the loop will fail when the `next` method is called for the second time.

Example 2

```
    ....
    PreparedStatement pstmt = conn.prepareStatement("SELECT first_name FROM
EMPLOYEES WHERE employee_id = ?");
    pstmt.setString(1, "205");
    ResultSet rs = pstmt.executeQuery();
    while (rs.next())
    {
        ....
    }

    Properties props = new Properties();
    props.put("PROXY_USER_NAME", proxyUser);

    conn.openProxySession(OracleConnection.PROXYTYPE_USER_NAME, props);
    .......
    conn.close(OracleConnection.PROXY_SESSION);

    // Try to use the PreparedStatement again
    pstmt.setString(1, "28960");
// This line of code will fail because the Statement is already closed while
closing the connection!
    rs = pstmt.executeQuery();
```

In the preceding example, the `PreparedStatement` object and the `ResultSet` object work fine before opening the proxy connection. But, if you try to execute the same `PreparedStatement` object after closing the proxy connection, then the statement fails.

Part IV

Data Access and Manipulation

This part provides a chapter that discusses about accessing and manipulating Oracle data. It also includes chapters that provide information about Java Database Connectivity (JDBC) support for user-defined object types, large object (LOB) and binary file (BFILE) locators and data, object references, and Oracle collections, such as nested tables. This part also provides chapters that discuss the result set functionality in JDBC, JDBC row sets, and globalization support provided by Oracle JDBC drivers.

Part IV contains the following chapters:

11

Accessing and Manipulating Oracle Data

This chapter describes Oracle extensions (`oracle.sql.*` formats) and compares them to standard Java formats (`java.sql.*`). Using Oracle extensions involves casting your result sets and statements to `OracleResultSet`, `OracleStatement`, `OraclePreparedStatement`, and `OracleCallableStatement`, as appropriate, and using the `getOracleObject`, `setOracleObject`, `get`*`XXX`*, and `set`*`XXX`* methods of these classes, where *`XXX`* corresponds to the types in the `oracle.sql` package.

This chapter covers the following topics:

- Data Type Mappings
- Data Conversion Considerations
- Result Set and Statement Extensions
- Comparison of Oracle get and set Methods to Standard JDBC
- Using Result Set Metadata Extensions
- Using SQL CALL and CALL INTO Statements

Data Type Mappings

The Oracle JDBC drivers support standard JDBC types as well as Oracle-specific data types. This section documents standard and Oracle-specific SQL-Java default type mappings. This section contains the following topics:

- Table of Mappings
- Notes Regarding Mappings

Table of Mappings

Table 11–1 shows the default mappings between SQL data types, JDBC type codes, standard Java types, and Oracle extended types.

The SQL Data Types column lists the SQL types that exist in Oracle Database 12*c* Release 1 (12.1). The JDBC Type Codes column lists data type codes supported by the JDBC standard and defined in the `java.sql.Types` class or by Oracle in the `oracle.jdbc.OracleTypes` class. For standard type codes, the codes are identical in these two classes.

The Standard Java Types column lists standard types defined in the Java language. The Oracle Extension Java Types column lists the `oracle.sql.*` Java types that correspond to each SQL data type in the database. These are Oracle extensions that let you retrieve all SQL data in the form of an `oracle.sql.*` Java type.

Note: In general, the Oracle JDBC drivers are optimized to manipulate SQL data using the standard JDBC types. In a few specialized cases, it may be advantageous to use the Oracle extension classes that are available in the `oracle.sql` package. But, Oracle strongly recommends to use the standard JDBC types instead of Oracle extensions, whenever possible. For more information about when to use Oracle extension, refer to "Standard Types Versus Oracle Types" on page 11-4.

See Also: "Package oracle.sql" on page 4-5 for more information on Oracle extensions

Table 11–1 Default Mappings Between SQL Types and Java Types

SQL Data Types	JDBC Type Codes	Standard Java Types	Oracle Extension Java Types
	STANDARD JDBC TYPES:		
CHAR	java.sql.Types.CHAR	java.lang.String	oracle.sql.CHAR
VARCHAR2	java.sql.Types.VARCHAR	java.lang.String	oracle.sql.CHAR
LONG	java.sql.Types.LONGVARCHAR	java.lang.String	oracle.sql.CHAR
NUMBER	java.sql.Types.NUMERIC	java.math.BigDecimal	oracle.sql.NUMBER
NUMBER	java.sql.Types.DECIMAL	java.math.BigDecimal	oracle.sql.NUMBER
NUMBER	java.sql.Types.BIT	boolean	oracle.sql.NUMBER
NUMBER	java.sql.Types.TINYINT	byte	oracle.sql.NUMBER
NUMBER	java.sql.Types.SMALLINT	short	oracle.sql.NUMBER
NUMBER	java.sql.Types.INTEGER	int	oracle.sql.NUMBER
NUMBER	java.sql.Types.BIGINT	long	oracle.sql.NUMBER
NUMBER	java.sql.Types.REAL	float	oracle.sql.NUMBER
NUMBER	java.sql.Types.FLOAT	double	oracle.sql.NUMBER
NUMBER	java.sql.Types.DOUBLE	double	oracle.sql.NUMBER
RAW	java.sql.Types.BINARY	byte[]	oracle.sql.RAW
RAW	java.sql.Types.VARBINARY	byte[]	oracle.sql.RAW
LONGRAW	java.sql.Types.LONGVARBINARY	byte[]	oracle.sql.RAW
DATE	java.sql.Types.DATE	java.sql.Date	oracle.sql.DATE
DATE	java.sql.Types.TIME	java.sql.Time	oracle.sql.DATE
TIMESTAMP	java.sql.Types.TIMESTAMP	javal.sql.Timestamp	oracle.sql.TIMESTAMP
BLOB	java.sql.Types.BLOB	java.sql.Blob	oracle.jdbc.OracleBlob[1]
CLOB	java.sql.Types.CLOB	java.sql.Clob	oracle.jdbc.OracleClob[2]
user-defined object	java.sql.Types.STRUCT	java.sql.Struct	oracle.jdbc.OracleStruct[3]
user-defined reference	java.sql.Types.REF	java.sql.Ref	oracle.jdbc.OracleRef[4]

Table 11–1 (Cont.) Default Mappings Between SQL Types and Java Types

SQL Data Types	JDBC Type Codes	Standard Java Types	Oracle Extension Java Types
user-defined collection	java.sql.Types.ARRAY	java.sql.Array	oracle.jdbc.OracleArray[5]
ROWID	java.sql.Types.ROWID	java.sql.RowId	oracle.sql.ROWID
NCLOB	java.sql.Types.NCLOB	java.sql.NClob	oracle.sql.NCLOB
NCHAR	java.sql.Types.NCHAR	java.lang.String	oracle.sql.CHAR
	ORACLE EXTENSIONS:		
BFILE	oracle.jdbc.OracleTypes.BFILE	NA	oracle.sql.BFILE
REF CURSOR	oracle.jdbc.OracleTypes.CURSOR	java.sql.ResultSet	oracle.jdbc.OracleResultSet
TIMESTAMP	oracle.jdbc.OracleTypes.TIMESTAMP	java.sql.Timestamp	oracle.sql.TIMESTAMP
TIMESTAMP WITH TIME ZONE	oracle.jdbc.OracleTypes.TIMESTAMPTZ	java.sql.Timestamp	oracle.sql.TIMESTAMPTZ
TIMESTAMP WITH LOCAL TIME ZONE	oracle.jdbc.OracleTypes.TIMESTAMPLTZ	java.sql.Timestamp	oracle.sql.TIMESTAMPLTZ

[1] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.BLOB` class is deprecated and replaced with the `oracle.jdbc.OracleBlob` interface.

[2] Starting from Oracle Database 12c Release 1, the `oracle.sql.CLOB` class is deprecated and is replaced with the `oracle.jdbc.OracleClob` interface.

[3] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.STRUCT` class is deprecated and replaced with the `oracle.jdbc.OracleStruct` interface.

[4] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.REF` class is deprecated and replaced with the `oracle.jdbc.OracleRef` interface.

[5] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.ARRAY` class is deprecated and replaced with the `oracle.jdbc.OracleArray` interface.

Note: For database versions, such as 8.1.7, which do not support the `TIMESTAMP` data type, `TIMESTAMP` is mapped to `DATE`.

See Also :

- "Supported SQL-JDBC Data Type Mappings" on page A-1
- Chapter 4, "Oracle Extensions"

Notes Regarding Mappings

This section provides further detail regarding mappings for `NUMBER` and user-defined types.

NUMBER Types

For the different type codes that an Oracle `NUMBER` value can correspond to, call the getter routine that is appropriate for the size of the data for mapping to work properly. For example, call `getByte` to get a Java `tinyint` value for an item x, where $-128 < x < 128$.

User-Defined Types

User-defined types, such as objects, object references, and collections, map by default to weak Java types, such as `java.sql.Struct`, but alternatively can map to strongly typed custom Java classes. Custom Java classes can implement one of two interfaces:

- The standard `java.sql.SQLData`
- The Oracle-specific `oracle.jdbc.OracleData`

> **See Also:** "Mapping Oracle Objects" on page 13-1 and "Creating and Using Custom Object Classes for Oracle Objects" on page 13-5

Data Conversion Considerations

When JDBC programs retrieve SQL data into Java, you can use standard Java types, or you can use types of the `oracle.sql` package. This section covers the following topics:

- Standard Types Versus Oracle Types
- Converting SQL NULL Data
- Testing for NULLs

Standard Types Versus Oracle Types

The Oracle data types in `oracle.sql` store data in the same bit format as used by the database. In versions of the Oracle JDBC drivers prior to Oracle Database 10*g*, the Oracle data types were generally more efficient. Starting from Oracle Database 10*g*, the JDBC drivers were substantially updated. As a result, in most cases the standard Java types are preferred to the data types in `oracle.sql.*`. In particular, `java.lang.String` is much more efficient than `oracle.sql.CHAR`.

In general, Oracle recommends that you use the Java standard types. The exceptions to this are:

- Use the `oracle.jdbc.OracleData` rather than the `java.sql.SqlData` if the `OracleData` functionality better suits your needs.
- Use `oracle.sql.NUMBER` rather than `java.lang.Double` if you need to retain the exact values of floating point numbers. Oracle `NUMBER` is a decimal representation and Java `Double` and `Float` are binary representations. Conversion from one format to the other can result in slight variations in the actual value represented. Additionally, the range of values that can be represented using the two formats is different.

  Use `oracle.sql.NUMBER` rather than `java.math.BigDecimal` when performance is critical and you are not manipulating the values, just reading and writing them.
- Use `oracle.sql.DATE` or `oracle.sql.TIMESTAMP` if you are using a JDK version earlier than JDK 6. Use `java.sql.Date` or `java.sql.Timestamp` if you are using JDK 6 or a later version.

  > **Note:** Due to a bug in all versions of Java prior to JDK 6, construction of `java.lang.Date` and `java.lang.Timestamp` objects is slow, especially in multithreaded environments. This bug is fixed in JDK 6.

- Use `oracle.sql.CHAR` only when you have data from some external source, which has been represented in an Oracle character set encoding. In all other cases, you should use `java.lang.String`.

- STRUCT, ARRAY, BLOB, CLOB, REF, and ROWID are all the implementation classes of the corresponding JDBC standard interface types. So, there is no benefit of using the Oracle extension types as they are identical to the JDBC standard types.
- BFILE, TIMESTAMPTZ, and TIMESTAMPLTZ have no representation in the JDBC standard. You must use these Oracle extensions.
- In all other cases, you should use the standard JDBC type rather than the Oracle extensions.

Note: If you convert an oracle.sql data type to a Java standard data type, then the benefits of using the oracle.sql data type are lost.

Converting SQL NULL Data

Java represents a SQL NULL datum by the Java value null. Java data types fall into two categories: primitive types, such as byte, int, and float, and object types, such as class instances. The primitive types cannot represent null. Instead, they store null as the value zero, as defined by the JDBC specification. This can lead to ambiguity when you try to interpret your results.

In contrast, Java object types can represent null. The Java language defines an object container type corresponding to every primitive type that can represent null. The object container types must be used as the targets for SQL data to detect SQL NULL without ambiguity.

Testing for NULLs

You cannot use a relational operator to compare NULL values with each other or with other values. For example, the following SELECT statement does not return any row even if the COMMISSION_PCT column contains one or more NULL values.

```
PreparedStatement pstmt = conn.prepareStatement(
  "SELECT * FROM EMPLOYEES WHERE COMMISSION_PCT = ?");
pstmt.setNull(1, java.sql.Types.VARCHAR);
```

The next example shows how to compare values for equality when some return values might be NULL. The following code returns all the FIRST_NAME from the EMPLOYEES table that are NULL, if there is no value of 205 for COMM.

```
PreparedStatement pstmt = conn.prepareStatement("SELECT FIRST_NAME FROM EMPLOYEES
  WHERE COMMISSION_PCT =? OR  ((COMM IS NULL) AND (? IS NULL))");
pstmt.setBigDecimal(1, new BigDecimal(205));
pstmt.setNull(2, java.sql.Types.VARCHAR);
```

Result Set and Statement Extensions

The Statement object returns a java.sql.ResultSet. If you want to apply only standard JDBC methods to the object, then keep it as a ResultSet type. However, if you want to use the Oracle extensions on the object, then you must cast it to OracleResultSet. All of the Oracle Result Set extensions are in the oracle.jdbc.OracleResultSet interface and all the Statement extensions are in the oracle.jdbc.OracleStatement interface.

For example, assuming you have a standard Statement object stmt, do the following if you want to use only standard JDBC ResultSet methods:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

If you need the extended functionality provided by the Oracle extensions to JDBC, you can select the results into a standard `ResultSet` variable and then cast that variable to `OracleResultSet` later.

Key extensions to the result set and statement classes include the `getOracleObject` and `setOracleObject` methods, used to access and manipulate data in `oracle.sql.*` formats.

Comparison of Oracle get and set Methods to Standard JDBC

This section describes `get` and `set` methods, particularly the JDBC standard `getObject` and `setObject` methods and the Oracle-specific `getOracleObject` and `setOracleObject` methods, and how to access data in `oracle.sql.*` format compared with Java format.

You can use the standard `get`*`XXX`* methods for all Oracle SQL types.

This section covers the following topics:

- Standard getObject Method
- Oracle getOracleObject Method
- Summary of getObject and getOracleObject Return Types
- Other getXXX Methods
- Data Types For Returned Objects from getObject and getXXX
- The setObject and setOracleObject Methods
- Other setXXX Methods

Note: You cannot qualify a column name with a table name and pass it as a parameter to the `get`*`XXX`* method. For example:

```
ResultSet rset = stmt.executeQuery("SELECT employees.department_id,
department.department_id FROM employees, department");
rset.getInt("employees.department_id");
```

The `getInt` method in the preceding code will throw an exception. To uniquely identify the columns in the `get`*`XXX`* method, you can either use column index or specify column aliases in the query and use these aliases in the `get`*`XXX`* method.

Standard getObject Method

The standard `getObject` method of a result set or callable statement has a return type of `java.lang.Object`. The class of the object returned is based on its SQL type, as follows:

- For SQL data types that are not Oracle-specific, the `getObject` method returns the default Java type corresponding to the SQL type of the column, following the mapping in the JDBC specification.
- For Oracle-specific data types, `getObject` returns an object of the appropriate `oracle.sql.*` class, such as `oracle.sql.ROWID`.

- For Oracle database objects, `getObject` returns a Java object of the class specified in your type map. Type maps specify a mapping from database named types to Java classes. The `getObject(parameter_index)` method uses the default type map of the connection. The `getObject(parameter_index, map)` enables you to pass in a type map. If the type map does not provide a mapping for a particular Oracle object, then `getObject` returns an `oracle.sql.OracleStruct` object.

Oracle getOracleObject Method

If you want to retrieve data from a result set or callable statement as an `oracle.sql.*` object, then you must follow a special process. For an `OracleResultSet` object, you must cast the Result Set to `oracle.jdbc.OracleResultSet` and then call `getOracleObject` instead of `getObject`. The same applies to `CallableStatement` and `oracle.jdbc.OracleCallableStatement`.

The return type of `getOracleObject` is `oracle.sql.Datum`. The actual returned object is an instance of the appropriate `oracle.sql.*` class. The method signature is:

```
public oracle.sql.Datum getOracleObject(int parameter_index)
```

When you retrieve data into a `Datum` variable, you can use the standard Java `instanceof` operator to determine which `oracle.sql.*` type it really is.

Example: Using getOracleObject with a Result Set

The following example creates a table that contains a column of `CHAR` data and a column containing a `BFILE` locator. A `SELECT` statement retrieves the contents of the table as a result set. The `getOracleObject` then retrieves the `CHAR` data into the `char_datum` variable and the `BFILE` locator into the `bfile_datum` variable. Note that because `getOracleObject` returns a `Datum` object, the return values must be cast to `CHAR` and `BFILE`, respectively.

```
stmt.execute ("CREATE TABLE bfile_table (x VARCHAR2 (30), b BFILE)");
stmt.execute
    ("INSERT INTO bfile_table VALUES ('one', BFILENAME ('TEST_DIR', 'file1'))");

ResultSet rset = stmt.executeQuery ("SELECT * FROM bfile_table");
while (rset.next ())
{
   CHAR char_datum = (CHAR) ((OracleResultSet)rset).getOracleObject (1);
   BFILE bfile_datum = (BFILE) ((OracleResultSet)rset).getOracleObject (2);
   ...
}
```

Example: Using getOracleObject in a Callable Statement

The following example prepares a call to the procedure `myGetDate`, which associates a character string with a date. The program passes `"HR"` to the prepared call and registers the `DATE` type as an output parameter. After the call is run, `getOracleObject` retrieves the date associated with `"HR"`. Note that because `getOracleObject` returns a `Datum` object, the results are cast to `DATE`.

```
OracleCallableStatement cstmt = (OracleCallableStatement)conn.prepareCall
                                    ("begin myGetDate (?, ?); end;");

cstmt.setString (1, "HR");
cstmt.registerOutParameter (2, Types.DATE);
cstmt.execute ();
```

```
DATE date = (DATE) ((OracleCallableStatement)cstmt).getOracleObject (2);
...
```

Summary of getObject and getOracleObject Return Types

Table 11–2 lists the underlying return types for the `getObject` and `getOracleObject` methods for each Oracle SQL type.

Keep in mind the following when you use these methods:

- `getObject` always returns data into a `java.lang.Object` instance
- `getOracleObject` always returns data into an `oracle.sql.Datum` instance

You must cast the returned object to use any special functionality.

Table 11–2 getObject and getOracleObject Return Types

Oracle SQL Type	getObject Underlying Return Type	getOracleObject Underlying Return Type
CHAR	String	oracle.sql.CHAR
VARCHAR2	String	oracle.sql.CHAR
NCHAR	String	oracle.sql.CHAR
LONG	String	oracle.sql.CHAR
NUMBER	java.math.BigDecimal	oracle.sql.NUMBER
RAW	byte[]	oracle.sql.RAW
LONGRAW	byte[]	oracle.sql.RAW
DATE	java.sql.Date	oracle.sql.DATE
TIMESTAMP	java.sql.Timestamp[1]	oracle.sql.TIMESTAMP
TIMESTAMP WITH TIME ZONE	oracle.sql.TIMESTAMPTZ	oracle.sql.TIMESTAMPTZ
TIMESTAMP WITH LOCAL TIME ZONE	oracle.sql.TIMESTAMPLTZ	oracle.sql.TIMESTAMPLTZ
BINARY_FLOAT	java.lang.Float	oracle.sql.BINARY_FLOAT
BINARY_DOUBLE	java.lang.Double	oracle.sql.BINARY_DOUBLE
INTERVAL DAY TO SECOND	oracle.sql.INTERVALDS	oracle.sql.INTERVALDS
INTERVAL YEAR TO MONTH	oracle.sql.INTERVALYM	oracle.sql.INTERVALYM
ROWID	oracle.sql.ROWID	oracle.sql.ROWID
REF CURSOR	java.sql.ResultSet	(not supported)
BLOB	oracle.jdbc.OracleBlob[2]	oracle.jdbc.OracleBlob
CLOB	oracle.jdbc.OracleClob[3]	oracle.jdbc.OracleClob
NCLOB	java.sql.NClob	oracle.sql.NCLOB
BFILE	oracle.sql.BFILE	oracle.sql.BFILE

Table 11–2 (Cont.) getObject and getOracleObject Return Types

Oracle SQL Type	getObject Underlying Return Type	getOracleObject Underlying Return Type
Oracle object	class specified in type map or `oracle.sql.OracleStruct`[4] (if no type map entry)	`oracle.jdbc.OracleStruct`
Oracle object reference	`oracle.jdbc.OracleRef`[5]	`oracle.jdbc.OracleRef`
collection (varray or nested table)	`oracle.jdbc.OracleArray`[6]	`oracle.sql.ARRAY`

[1] `ResultSet.getObject` returns `java.sql.Timestamp` only if the `oracle.jdbc.J2EE13Compliant` connection property is set to `TRUE`, else the method returns `oracle.sql.TIMESTAMP`.

[2] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.BLOB` class is deprecated and replaced with the `oracle.jdbc.OracleBlob` interface.

[3] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.CLOB` class is deprecated and replaced with the `oracle.jdbc.OracleClob` interface.

[4] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.STRUCT` class is deprecated and replaced with the `oracle.jdbc.OracleStruct` interface.

[5] Starting from Oracle Database 12*c* Release 1, the `oracle.sql.REF` class is deprecated and is replaced with the `oracle.jdbc.OracleRef` interface.

[6] Starting from Oracle Database 12*c* Release 1, the `oracle.sql.ARRAY` class is deprecated and replaced with the `oracle.jdbc.OracleArray` interface.

Note: The `ResultSet.getObject` method returns `java.sql.Timestamp` for the `TIMESTAMP` SQL type, only when the connection property `oracle.jdbc.J2EE13Compliant` is set to `TRUE`. This property has to be set when the connection is obtained. If this connection property is not set or if it is set after the connection is obtained, then the `ResultSet.getObject` method returns `oracle.sql.TIMESTAMP` for the `TIMESTAMP` SQL type.

The `oracle.jdbc.J2EE13Compliant` connection property can also be set without changing the code in the following ways:

- Including the `ojdbc6dms.jar` or `ojdbc7dms.jar` files in the `CLASSPATH`. These files set `oracle.jdbc.J2EE13Compliant` to `TRUE` by default. These are specific to the Oracle Application Server release and are not available as part of the general JDBC release. They are located in `$ORACLE_HOME/jdbc/lib`.
- Setting the system property by calling the `java` command with the flag `-Doracle.jdbc.J2EE13Compliant=true`. For example,

```
java -Doracle.jdbc.J2EE13Compliant=true ...
```

When the `J2EE13Compliant` is set to `TRUE` the action is as in Table B-3 of the JDBC specification.

See Also: Table A–1, " Valid SQL Data Type-Java Class Mappings" on page A-1, for information about type compatibility between all SQL and Java types.

Other getXXX Methods

Standard JDBC provides a `get`*`XXX`* for each standard Java type, such as `getByte`, `getInt`, `getFloat`, and so on. Each of these returns exactly what the method name implies.

In addition, the `OracleResultSet` and `OracleCallableStatement` interfaces provide a full complement of `get`*`XXX`* methods corresponding to all the `oracle.sql.*` types. Each `get`*`XXX`* method returns an `oracle.sql.`*`XXX`* object. For example, `getROWID` returns an `oracle.sql.ROWID` object.

There is no performance advantage in using the specific `get`*`XXX`* methods. However, they do save you the trouble of casting, because the return type is specific to the object being returned.

This section covers the following topics:

- Return Types of getXXX Methods
- Special Notes about getXXX Methods

Return Types of getXXX Methods

Refer to the JDBC Javadoc to know the return types for each `get`*`XXX`* method and also which are Oracle extensions under Java Development Kit (JDK) 6. You must cast the returned object to `OracleResultSet` or `OracleCallableStatement` to use methods that are Oracle extensions.

Special Notes about getXXX Methods

This section provides additional details about some `get`*`XXX`* methods.

getBigDecimal

JDBC 2.0 simplified method signatures for the `getBigDecimal` method. The previous input signatures were:

```
(int columnIndex, int scale) or (String columnName, int scale)
```

The simplified input signature is:

```
(int columnIndex) or (String columnName)
```

The `scale` parameter, used to specify the number of digits to the right of the decimal, is no longer necessary. The Oracle JDBC drivers retrieve numeric values with full precision.

getBoolean

Because there is no `BOOLEAN` database type, when you use `getBoolean` a data type conversion always occurs. The `getBoolean` method is supported only for numeric columns. When applied to these columns, `getBoolean` interprets any zero value as `false` and any other value as `true`. When applied to any other sort of column, `getBoolean` raises the exception `java.lang.NumberFormatException`.

Data Types For Returned Objects from getObject and getXXX

The return type of `getObject` is `java.lang.Object`. The returned value is an instance of a subclass of `java.lang.Object`. Similarly, the return type of `getOracleObject` is `oracle.sql.Datum`, and the class of the returned value is a subclass of `oracle.sql.Datum`. You typically cast the returned object to the appropriate class to use particular methods and functionality of that class.

In addition, you have the option of using a specific `get`*`XXX`* method instead of the generic `getObject` or `getOracleObject` methods. The `get`*`XXX`* methods enable you to avoid casting, because the return type of `get`*`XXX`* corresponds to the type of object returned. For example, the return type of `getCLOB` is `oracle.sql.CLOB`, as opposed to `java.lang.Object`.

Example of Casting Return Values

This example assumes that you have fetched data of the `NUMBER` type as the first column of a result set. Because you want to manipulate the `NUMBER` data without losing precision, cast your result set to `OracleResultSet` and use `getOracleObject` to return the `NUMBER` data in `oracle.sql.*` format. If you do not cast your result set, then you have to use `getObject`, which returns your numeric data into a Java `Float` and loses some of the precision of your SQL data.

The `getOracleObject` method returns an `oracle.sql.NUMBER` object into an `oracle.sql.Datum` return variable unless you cast the output. Cast the `getOracleObject` output to `oracle.sql.NUMBER` if you want to use a `NUMBER` return variable and any of the special functionality of that class.

```
NUMBER x = (NUMBER)ors.getOracleObject(1);
```

The setObject and setOracleObject Methods

Just as there is a standard `getObject` and Oracle-specific `getOracleObject` in result sets and callable statements, there are also standard `setObject` and Oracle-specific `setOracleObject` methods in `OraclePreparedStatement` and `OracleCallableStatement`. The `setOracleObject` methods take `oracle.sql.*` input parameters.

To bind standard Java types to a prepared statement or callable statement, use the `setObject` method, which takes a `java.lang.Object` as input. The `setObject` method does support a few of the `oracle.sql.*` types. However, the method has been implemented so that you can enter instances of the `oracle.sql.*` classes that correspond to the following JDBC standard types: `Blob`, `Clob`, `Struct`, `Ref`, and `Array`.

To bind `oracle.sql.*` types to a prepared statement or callable statement, use the `setOracleObject` method, which takes a subclass of `oracle.sql.Datum` as input. To use `setOracleObject`, you must cast your prepared statement or callable statement to `OraclePreparedStatement` or `OracleCallableStatement`.

Example of Using setObject and setOracleObject

For a prepared statement, the `setOracleObject` method binds the `oracle.sql.CHAR` data represented by the `charVal` variable to the prepared statement. To bind the `oracle.sql.*` data, the prepared statement must be cast to `OraclePreparedStatement`. Similarly, the `setObject` method binds the Java `String` data represented by the variable `strVal`.

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
((OraclePreparedStatement)ps).setOracleObject(1,charVal);
ps.setObject(2,strVal);
```

Other setXXX Methods

As with the `get`*`XXX`* methods, there are several specific `set`*`XXX`* methods. Standard `set`*`XXX`* methods are provided for binding standard Java types, and Oracle-specific `set`*`XXX`* methods are provided for binding Oracle-specific types.

Similarly, there are two forms of the `setNull` method:

- `void setNull(int parameterIndex, int sqlType)`

  This is specified in the standard `java.sql.PreparedStatement` interface. This signature takes a parameter index and a SQL type code defined by the `java.sql.Types` or `oracle.jdbc.OracleTypes` class. Use this signature to set an object other than a `REF`, `ARRAY`, or `STRUCT` to `NULL`.

- `void setNull(int parameterIndex, int sqlType, String sql_type_name)`

  With JDBC 2.0, this signature is also specified in the standard `java.sql.PreparedStatement` interface. This method takes a SQL type name in addition to a parameter index and a SQL type code. Use this method when the SQL type code is `java.sql.Types.REF`, `ARRAY`, or `STRUCT`. If the type code is other than `REF`, `ARRAY`, or `STRUCT`, then the given SQL type name is ignored.

Similarly, the `registerOutParameter` method has a signature for use with `REF`, `ARRAY`, or `STRUCT` data:

```
void registerOutParameter
            (int parameterIndex, int sqlType, String sql_type_name)
```

Binding Oracle-specific types using the appropriate `setXXX` methods, instead of the methods used for binding standard Java types, may offer some performance advantage.

This section covers the following topics:

- Input Data Binding
- Method setFixedCHAR for Binding CHAR Data into WHERE Clauses

Input Data Binding

There are three way to bind data for input:

- Direct binding where the data itself is placed in a bind buffer
- Stream binding where the data is streamed
- LOB binding where a temporary lob is created, the data placed in the LOB using the LOB APIs, and the bytes of the LOB locator are placed in the bind buffer

The three kinds of binding have some differences in performance and have an impact on batching. Direct binding is fast and batching is fine. Stream binding is slower, may require multiple round trips, and turns batching off. LOB binding is very slow and requires many round trips. Batching works, but might be a bad idea. They also have different size limits, depending on the type of the SQL statement.

For SQL parameters, the length of standard parameter types, such as `RAW` and `VARCHAR2`, is fixed by the size of the target column. For PL/SQL parameters, the size is limited to a fixed number of bytes, which is 32766.

In Oracle Database 10*g* release 2, certain changes were made to the `setString`, `setCharacterStream`, `setAsciiStream`, `setBytes`, and `setBinaryStream` methods of `PreparedStatement`. The original behavior of these APIs were:

- `setString`: Direct bind of characters
- `setCharacterStream`: Stream bind of characters
- `setAsciiStream`: Stream bind of bytes
- `setBytes`: Direct bind of bytes

- `setBinaryStream`: Stream bind of bytes

Starting from Oracle Database 10*g* Release 2, automatic switching between binding modes, based on the data size and on the type of the SQL statement is provided.

setBytes and setBinaryStream

For SQL, direct bind is used for size up to 2000 and stream bind for larger.

For PL/SQL direct bind is used for size up to 32766 and LOB bind is used for larger.

setString, setCharacterStream, and setAsciiStream

For SQL, direct bind is used up to 32766 Java characters and stream bind is used for larger. This is independent of character set.

For PL/SQL, you must be careful about the byte size of the character data in the database character set or the national character set depending on the setting of the form of use parameter. Direct bind is used for data where the byte length is less than 32766 and LOB bind is used for larger.

For fixed length character sets, multiply the length of the Java character data by the fixed character size in bytes and compare that to the restrictive values. For variable length character sets, there are three cases based on the Java character length, as follows:

- If character length is less than 32766 divided by the maximum character size, then direct bind is used.
- If character length is greater than 32766 divided by the minimum character size, then LOB bind is used.
- If character length is in between and if the actual length of the converted bytes is less than 32766, then direct bind is used, else LOB bind is used.

> **Note:** When a PL/SQL procedure is embedded in a SQL statement, the binding action is different. Refer to "Data Interface for LOBs" on page 14-3 for more information.

The server-side internal driver has the following additional limitations:

- `setString`, `setCharacterStream`, and `setASCIIStream` APIs are not supported for SQL CLOB columns when the data size in characters is over 32767 bytes
- `setBytes` and `setBinaryStream` APIs are not supported for SQL BLOB columns when the data size is over 32767 bytes

> **Important:** Do not use these APIs with the server-side internal driver, without careful checking of the data size in client code.

> **See Also:** JDBC Release Notes for further discussion and possible workarounds

Method setFixedCHAR for Binding CHAR Data into WHERE Clauses

`CHAR` data in the database is padded to the column width. This leads to a limitation in using the `setCHAR` method to bind character data into the `WHERE` clause of a `SELECT` statement. The character data in the `WHERE` clause must also be padded to the column

width to produce a match in the SELECT statement. This is especially troublesome if you do not know the column width.

To remedy this, Oracle has added the setFixedCHAR method to the OraclePreparedStatement class. This method runs a non-padded comparison.

Note:

- Remember to cast your prepared statement object to OraclePreparedStatement to use the setFixedCHAR method.
- There is no need to use setFixedCHAR for an INSERT statement. The database always automatically pads the data to the column width as it inserts it.

Example

The following example demonstrates the difference between the setCHAR and setFixedCHAR methods.

```
/* Schema is :
 create table my_table (col1 char(10));
 insert into my_table values ('JDBC');
*/
 PreparedStatement pstmt = conn.prepareStatement
                   ("select count(*) from my_table where col1 = ?");

 pstmt.setString (1, "JDBC");  // Set the Bind Value
 runQuery (pstmt);             // This will print " No of rows are 0"

 CHAR ch = new CHAR("JDBC      ", null);
 ((OraclePreparedStatement)pstmt).setCHAR(1, ch); // Pad it to 10 bytes
 runQuery (pstmt);             // This will print "No of rows are 1"

 ((OraclePreparedStatement)pstmt).setFixedCHAR(1, "JDBC");
  runQuery (pstmt);            // This will print "No of rows are 1"

 void runQuery (PreparedStatement ps)
 {
   // Run the Query
   ResultSet rs = pstmt.executeQuery ();

   while (rs.next())
     System.out.println("No of rows are " + rs.getInt(1));

   rs.close();
   rs = null;
 }
```

Using Result Set Metadata Extensions

The oracle.jdbc.OracleResultSetMetaData interface is JDBC 2.0-compliant but does not implement the getSchemaName and getTableName methods because Oracle Database does not make this feasible.

The following code snippet uses several of the methods in the OracleResultSetMetadata interface to retrieve the number of columns from the EMPLOYEES table and the numerical type and SQL type name of each column:

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rset = dbmd.getTables("", "HR", "EMPLOYEES", null);

 while (rset.next())
 {
   OracleResultSetMetaData orsmd = ((OracleResultSet)rset).getMetaData();
   int numColumns = orsmd.getColumnCount();
   System.out.println("Num of columns = " + numColumns);

   for (int i=0; i<numColumns; i++)
   {
     System.out.print ("Column Name=" + orsmd.getColumnName (i+1));
     System.out.print (" Type=" + orsmd.getColumnType (i + 1) );
     System.out.println (" Type Name=" + orsmd.getColumnTypeName (i + 1));
  }
}
```

The program returns the following output:

```
Num of columns = 5
Column Name=TABLE_CAT Type=12 Type Name=VARCHAR2
Column Name=TABLE_SCHEM Type=12 Type Name=VARCHAR2
Column Name=TABLE_NAME Type=12 Type Name=VARCHAR2
Column Name=TABLE_TYPE Type=12 Type Name=VARCHAR2
Column Name=TABLE_REMARKS Type=12 Type Name=VARCHAR2
```

Using SQL CALL and CALL INTO Statements

You can use the CALL statement to execute a routine from within SQL.

> **Note:** A routine is a procedure or a function that is standalone or is defined within a type or package. You must have `EXECUTE` privilege on the standalone routine or on the type or package in which the routine is defined. Refer to the "*Oracle Database SQL Language Reference*" for more information about using the `CALL` statement.

You can execute a routine in two ways:

- By issuing a call to the routine itself by name or by using the `routine_clause`
- By using an `object_access_expression` inside the type of an expression

You can specify one or more arguments to the routine, if the routine takes arguments. You can use positional, named, or mixed notation for argument.

CALL INTO Statement

The `INTO` clause applies only to calls to functions. You can use the following types of variables with this clause:

- Host variable
- Indicator variable

PL/SQL Blocks

The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other. A PL/SQL block has three parts: a declarative

part, an executable part, and an exception-handling part. You get the following advantages by using PL/SQL blocks in your application:

- Better performance
- Higher productivity
- Full portability
- Tight integration with Oracle
- Tight security

12
Java Streams in JDBC

This chapter describes how the Oracle Java Database Connectivity (JDBC) drivers handle Java streams for several data types. Data streams enable you to read `LONG` column data of up to 2 gigabytes (GB).

This chapter covers the following topics:

Overview of Java Streams

Oracle JDBC drivers support the manipulation of data streams in either direction between server and client. The drivers support all stream conversions: binary, ASCII, and Unicode. Following is a brief description of each type of stream:

- Binary

  Used for `RAW` bytes of data, and corresponds to the `getBinaryStream` method

- ASCII

  Used for ASCII bytes in ISO-Latin-1 encoding, and corresponds to the `getAsciiStream` method

- Unicode

  Used for Unicode bytes with the `UTF-16` encoding, and corresponds to the `getUnicodeStream` method

The `getBinaryStream`, `getAsciiStream`, and `getUnicodeStream` methods return the bytes of data in an `InputStream` object.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), the `CONNECTION_PROPERTY_STREAM_CHUNK_SIZE` is deprecated and the driver does not use it internally for setting the stream chunk size.

See Also: Chapter 14, "Working with LOBs and BFILEs"

Streaming LONG or LONG RAW Columns

When a query selects one or more `LONG` or `LONG RAW` columns, the JDBC driver transfers these columns to the client in streaming mode. In streaming mode, the JDBC driver does not read the column data from the network for `LONG` or `LONG RAW` columns, until required. The column data remains in the network communications channel until your code calls a `getXXX` method to read the column data. Even after the call, the column data is read only as needed to populate return value from the getXXX call. Because the column data remains in the communications channel, the streaming mode interferes with all other use of the connection. Any use of the connection, other than reading the column data, will discard the column data from the channel. While the streaming mode makes efficient use of memory and minimizes network round trips, it interferes with many other database operations.

Note: Oracle recommends avoiding `LONG` and `LONG RAW` columns. Use `LOB` instead.

To access the data in a `LONG` column, you can get the column as a Java `InputStream` object and use the `read` method of the `InputStream` object. As an alternative, you can get the data as a `String` or `byte` array. In this case, the driver will do the streaming for you.

You can get `LONG` and `LONG RAW` data with any of the three stream types. The driver performs conversions for you, depending on the character set of the database and the driver.

Note: Do not create tables with `LONG` columns. Use large object (LOB) columns, `CLOB`, `NCLOB`, and `BLOB`, instead. `LONG` columns are supported only for backward compatibility. Oracle recommends that you convert existing `LONG` columns to `LOB` columns. `LOB` columns are subject to far fewer restrictions than `LONG` columns.

This section covers the following topics:

- Streaming LONG or LONG RAW Columns
- Streaming CHAR, VARCHAR, or RAW Columns
- Streaming LOBs and External Files
- Data Streaming and Multiple Columns
- Closing a Stream
- Notes and Precautions on Streams

LONG RAW Data Conversions

A call to `getBinaryStream` returns `RAW` data. A call to `getAsciiStream` converts the `RAW` data to hexadecimal and returns the ASCII representation. A call to `getUnicodeStream` converts the `RAW` data to hexadecimal and returns the Unicode characters.

LONG Data Conversions

When you get LONG data with getAsciiStream, the drivers assume that the underlying data in the database uses an US7ASCII or WE8ISO8859P1 character set. If the assumption is true, then the drivers return bytes corresponding to ASCII characters. If the database is not using an US7ASCII or WE8ISO8859P1 character set, a call to getAsciiStream returns meaningless information.

When you get LONG data with getUnicodeStream, you get a stream of Unicode characters in the UTF-16 encoding. This applies to all underlying database character sets that Oracle supports.

When you get LONG data with getBinaryStream, there are two possible cases:

- If the driver is JDBC OCI and the *client* character set is *not* US7ASCII or WE8ISO8859P1, then a call to getBinaryStream returns UTF-8. If the *client* character set is US7ASCII or WE8ISO8859P1, then the call returns a US7ASCII stream of bytes.
- If the driver is JDBC Thin and the *database* character set is *not* US7ASCII or WE8ISO8859P1, then a call to getBinaryStream returns UTF-8. If the server-side character set is US7ASCII or WE8ISO8859P1, then the call returns a US7ASCII stream of bytes.

See Also: Chapter 19, "Globalization Support"

Note: Receiving LONG or LONG RAW columns as a stream requires you to pay special attention to the order in which you retrieve columns from the database. For more information, see "Data Streaming and Multiple Columns" on page 12-7.

Table 12–1 summarizes LONG and LONG RAW data conversions for each stream type.

Table 12–1 LONG and LONG RAW Data Conversions

Data type	BinaryStream	AsciiStream	UnicodeStream
LONG	Bytes representing characters in Unicode UTF-8. The bytes can represent characters in US7ASCII or WE8ISO8859P1 if the database character set is US7ASCII or WE8ISO8859P1.	Bytes representing characters in ISO-Latin-1 (WE8ISO8859P1) encoding	Bytes representing characters in Unicode UTF-16 encoding
LONG RAW	unchanged data	ASCII representation of hexadecimal bytes	Unicode representation of hexadecimal bytes

Streaming Example for LONG RAW Data

One of the features of a getXXXStream method is that it enables you to fetch data incrementally. In contrast, getBytes fetches all the data in one call. This section contains two examples of getting a stream of binary data. The first version uses the getBinaryStream method to obtain LONG RAW data, and the second version uses the getBytes method.

Getting a LONG RAW Data Column with getBinaryStream

This example writes the contents of a LONG RAW column to a file on the local file system. In this case, the driver fetches the data incrementally.

The following code creates the table that stores a column of LONG RAW data associated with the name LESLIE:

```
-- SQL code:
create table streamexample (NAME varchar2 (256), GIFDATA long raw);
insert into streamexample values ('LESLIE', '00010203040506070809');
```

The following Java code snippet writes the data from the LONG RAW column into a file called leslie.gif:

```
ResultSet rset = stmt.executeQuery
                 ("select GIFDATA from streamexample where NAME='LESLIE'");

// get first row
if (rset.next())
{
    // Get the GIF data as a stream from Oracle to the client
    InputStream gif_data = rset.getBinaryStream (1);
   try
   {
      FileOutputStream file = null;
      file = new FileOutputStream ("leslie.gif");
      int chunk;
      while ((chunk = gif_data.read()) != -1)
         file.write(chunk);
   }
   catch (Exception e)
   {
      String err = e.toString();
      System.out.println(err);
   }
   finally
   {
      if file != null()
         file.close();
   }
}
```

In this example, the InputStream object returned by the call to getBinaryStream reads the data directly from the database connection.

Getting a LONG RAW Data Column with getBytes

This example gets the content of the GIFDATA column with getBytes instead of getBinaryStream. In this case, the driver fetches all the data in one call and stores it in a byte array. The code snippet is as follows:

```
ResultSet rset2 = stmt.executeQuery
                  ("select GIFDATA from streamexample where NAME='LESLIE'");

// get first row
if (rset2.next())
{
   // Get the GIF data as a stream from Oracle to the client
   byte[] bytes = rset2.getBytes(1);
   try
   {
      FileOutputStream file = null;
      file = new FileOutputStream ("leslie2.gif");
      file.write(bytes);
   }
```

```
   catch (Exception e)
   {
      String err = e.toString();
      System.out.println(err);
   }
   finally
   {
      if file != null()
         file.close();
   }
}
```

Because a LONG RAW column can contain up to 2 gigabytes of data, the getBytes example can use much more memory than the getBinaryStream example. Use streams if you do not know the maximum size of the data in your LONG or LONG RAW columns.

Avoiding Streaming for LONG or LONG RAW

> **Note:** Starting from Oracle Database 12*c* Release 1 (12.1), this method is deprecated. For more information, refer to "Deprecated Features" on page 3-xxxi.

The JDBC driver automatically streams any LONG and LONG RAW columns. However, there may be situations where you want to avoid data streaming. For example, if you have a very small LONG column, then you may want to avoid returning the data incrementally and, instead, return the data in one call.

To avoid streaming, use the defineColumnType method to redefine the type of the LONG column. For example, if you redefine the LONG or LONG RAW column as VARCHAR or VARBINARY type, then the driver will not automatically stream the data.

If you redefine column types with defineColumnType, then you must declare the types of the columns in the query. If you do not declare the types of the columns, then executeQuery will fail. In addition, you must cast the Statement object to oracle.jdbc.OracleStatement.

As an added benefit, using defineColumnType saves the OCI driver a database round-trip when running the query. Without defineColumnType, these JDBC drivers must request the data types of the column types. The JDBC Thin driver derives no benefit from defineColumnType, because it always uses the minimum number of round-trips.

Using the example from the previous section, the Statement object stmt is cast to OracleStatement and the column containing LONG RAW data is redefined to be of the type VARBINARAY. The data is not streamed. Instead, it is returned in a byte array. The code snippet is as follows:

```
//cast the statement stmt to an OracleStatement
oracle.jdbc.OracleStatement ostmt =
   (oracle.jdbc.OracleStatement)stmt;

//redefine the LONG column at index position 1 to VARBINARY
ostmt.defineColumnType(1, Types.VARBINARY);

// Do a query to get the images named 'LESLIE'
ResultSet rset = ostmt.executeQuery
         ("select GIFDATA from streamexample where NAME='LESLIE'");
```

```
// The data is not streamed here
rset.next();
byte [] bytes = rset.getBytes(1);
```

Streaming CHAR, VARCHAR, or RAW Columns

Note: Starting from Oracle Database 12*c* Release 1 (12.1), this method is deprecated. For more information, refer to "Deprecated Features" on page 3-xxxi.

If you use the `defineColumnType` Oracle extension to redefine a `CHAR`, `VARCHAR`, or `RAW` column as a `LONGVARCHAR` or `LONGVARBINARY`, then you can get the column as a stream. The program will behave as if the column were actually of type `LONG` or `LONG RAW`. Note that there is not much point to this, because these columns are usually short.

If you try to get a `CHAR`, `VARCHAR`, or `RAW` column as a data stream without redefining the column type, then the JDBC driver will return a Java `InputStream`, but no real streaming occurs. In the case of these data types, the JDBC driver fully fetches the data into an in-memory buffer during a call to the `executeQuery` method or the `next` method. The `getXXXStream` entry points return a stream that reads data from this buffer.

Streaming LOBs and External Files

The term large object (LOB) refers to a data item that is too large to be stored directly in a database table. Instead, a locator is stored in the database table, which points to the location of the actual data. External files are managed similarly. The JDBC drivers can support the following types through the use of streams:

- Binary large object (BLOB)

  For unstructured binary data
- Character large object (CLOB)

  For character data
- National Character large object (NCLOB)

  For national character data
- Binary file (BFILE)

  For external files

LOBs and BFILEs behave differently from the other types of streaming data described in this chapter. Instead of storing the actual data in the table, a locator is stored. The actual data can be manipulated using this locator, including reading and writing the data as a stream. Even when streaming, only the chunk of data (defined by a size) is streamed across the network. By contrast, when streaming a `LONG` or `LONG RAW`, the entire data is streamed across the network.

Streaming BLOBs, CLOBs, and NCLOBs

When a query fetches one or more `BLOB`, `CLOB`, or `NCLOB` columns, the JDBC driver transfers the data to the client. This data can be accessed as a stream. To manipulate `BLOB`, `CLOB`, or `NCLOB` data from JDBC, use methods in the Oracle extension classes `oracle.sql.BLOB`, `oracle.sql.CLOB` and `oracle.sql.NCLOB`. These classes provide

specific functionality, such as reading from the BLOB, CLOB, or NCLOB into an input stream, writing from an output stream into a BLOB, CLOB, or NCLOB, determining the length of a BLOB, CLOB, or NCLOB, and closing a BLOB, CLOB, or NCLOB.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), the concrete classes in the oracle.sql package are deprecated and replaced with the interfaces in the oracle.jdbc package. Oracle recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about these interfaces.

See Also: "Data Interface for LOBs" on page 14-3

Streaming BFILEs

An external file, or BFILE, is used to store a locator to a file outside the database. The file can be stored somewhere on the file system of the data server. The locator points to the actual location of the file.

When a query fetches one or more BFILE columns, the JDBC driver transfers the file to the client as required. The data can be accessed as a stream To manipulate BFILE data from JDBC, use methods in the Oracle extension class oracle.sql.BFILE. This class provides specific functionality, such as reading from the BFILE into an input stream, writing from an output stream into a BFILE, determining the length of a BFILE, and closing a BFILE.

Data Streaming and Multiple Columns

If a query fetches multiple columns and one of the columns contains a data stream, then the contents of the columns following the stream column are not available until the stream has been read, and the stream column is no longer available once any following column is read. Any attempt to read a column beyond a streaming column closes the streaming column.

See Also: "Streaming Data Precautions" on page 12-9

Streaming Example with Multiple Columns

Consider the following code:

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   //get the date data
   java.sql.Date date = rset.getDate(1);

   // get the streaming data
   InputStream is = rset.getAsciiStream(2);

   // Open a file to store the gif data
   FileOutputStream file = new FileOutputStream ("ascii.dat");

   // Loop, reading from the ascii stream and
   // write to the file
   int chunk;
```

```
   while ((chunk = is.read ()) != -1)
      file.write(chunk);
   // Close the file
   file.close();

   //get the number column data
   int n = rset.getInt(3);
}
```

The incoming data for each row has the following shape:

```
<a date><the characters of the long column><a number>
```

As you process each row of the result set, you must complete any processing of the stream column before reading the number column.

See Also: "Streaming LOBs and External Files" on page 12-6

Bypassing Streaming Data Columns

There may be situations where you want to avoid reading a column that contains streaming data. If you do not want to read such data, then call the `close` method of the stream object. This method discards the stream data and enables the driver to continue reading data from all the columns that contain non-streaming data and follow the column containing streaming data. Even though you are intentionally discarding the stream, it is a good programming practice to retrieve the columns in the same order as in the `SELECT` statement.

In the following example, the stream data in the `LONG` column is discarded and the data from only the `DATE` and `NUMBER` column is recovered:

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");

while rset.next()
{
   //get the date
   java.sql.Date date = rset.getDate(1);

   // access the stream data and discard it with close()
   InputStream is = rset.getAsciiStream(2);
   is.close();

   // get the number column data
   int n = rset.getInt(3);
}
```

Closing a Stream

You can discard the data from a stream at any time by calling the `close` method. It is a good programming practice to close the stream when you no longer need it. For example:

```
   ...
   InputStream is = rset.getAsciiStream(2);
   is.close();
```

See Also: "Bypassing Streaming Data Columns" on page 12-8 and "Streaming Data Precautions" on page 12-9

Note: Closing a stream has little performance effect on a LONG or LONG RAW column. All of the data still move across the network and the driver must read the bits from the network.

Notes and Precautions on Streams

This section discusses several cautionary issues regarding the use of streams:

- Streaming Data Precautions
- Using Streams to Avoid Limits on setBytes and setString
- Streaming and Row Prefetching

Streaming Data Precautions

This section describes some of the precautions you must take to ensure that you do not accidentally discard or lose your stream data. The drivers automatically discard stream data if you perform any JDBC operation that communicates with the database, other than reading the current stream. Two common precautions are:

- Use the stream data after you access it.

  To recover the data from a column containing a data stream, it is not enough to fetch the column. You must immediately process the contents of the column. Otherwise, the contents will be discarded when you fetch the next column.

- Call the stream column in the same order as in the `SELECT` statement.

  If your query fetches multiple columns, the database sends each row as a set of bytes representing the columns in the `SELECT` order. If one of the columns contains stream data, then the database sends the entire data stream before proceeding to the next column.

  If you do not use the order as in the `SELECT` statement to access data, then you can lose the stream data. That is, if you bypass the stream data column and access data in a column that follows it, then the stream data will be lost. For example, if you try to access the data for the `NUMBER` column *before* reading the data from the stream data column, then the JDBC driver first reads then discards the streaming data automatically. This can be very inefficient if the `LONG` column contains a large amount of data.

  If you try to access the `LONG` column later in the program, then the data will not be available and the driver will return a "`Stream Closed`" error.

The later point is illustrated in the following example:

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   int n = rset.getInt(3);  // This discards the streaming data
   InputStream is = rset.getAsciiStream(2);
                            // Raises an error: stream closed.
}
```

If you get the stream but do not use it *before* you get the `NUMBER` column, then the stream still closes automatically:

```
ResultSet rset = stmt.executeQuery
                 ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
```

```
while rset.next()
{
   InputStream is = rset.getAsciiStream(2); // Get the stream
   int n = rset.getInt(3);
   // Discards streaming data and closes the stream
}
int c = is.read(); // c is -1: no more characters to read-stream closed
```

Using Streams to Avoid Limits on setBytes and setString

Starting from Oracle Database 12*c*, the size limit of the data that is used with the `setBytes` and `setString` methods, have been increased significantly. Any Java `byte` array can be passed to `setBytes`, and any Java `String` can be passed to `setString`. The JDBC driver automatically switches to using `setBinaryStream` or `setCharacterStream` or to using `setBytesForBlob` or `setStringForClob`, depending on the size of the data, whether the statement is SQL or PL/SQL, and the driver used.

There are some limitation with earlier versions of Oracle Database and in the server-side internal driver.

> **See Also:** "Data Interface for LOBs" on page 14-3 and release notes for details

Streaming and Row Prefetching

If the JDBC driver encounters a column containing a data stream, then row fetch size is set back to one. Row fetch size is an Oracle performance enhancement that enables multiple rows of data to be retrieved with each trip to the database.

13

Working with Oracle Object Types

This chapter describes the Java Database Connectivity (JDBC) support for user-defined object types. It discusses functionality of the generic, weakly typed `oracle.sql.STRUCT` class, as well as how to map to custom Java classes that implement either the JDBC standard `SQLData` interface or the Oracle-specific `OracleData` interface.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.STRUCT` class is deprecated and replaced with the `oracle.jdbc.OracleStruct` interface, which is a part of the `oracle.jdbc` package. Oracle strongly recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleStruct` interface.

See Also: "Using PL/SQL Types" on page A-6

The following topics are covered:

- Mapping Oracle Objects
- Using the Default STRUCT Class for Oracle Objects
- Creating and Using Custom Object Classes for Oracle Objects
- Object-Type Inheritance
- Using JPublisher to Create Custom Object Classes
- Describing an Object Type

See Also: *Oracle Database Object-Relational Developer's Guide*

Mapping Oracle Objects

Oracle object types provide support for composite data structures in the database. For example, you can define a `Person` type that has the attributes `name` of `CHAR` type, `phoneNumber` of `CHAR` type, and `employeeNumber` of `NUMBER` type.

Oracle provides tight integration between its Oracle object features and its JDBC functionality. You can use a standard, generic JDBC type to map to Oracle objects, or you can customize the mapping by creating custom Java type definition classes.

> **Note:** In this book, Java classes that you create to map to Oracle objects will be referred to as **custom Java classes** or, more specifically, **custom object classes**. This is as opposed to **custom references classes**, which are Java classes that map to object references, and **custom collection classes**, which are Java classes that map to Oracle collections.

Custom object classes can implement either a standard JDBC interface or an Oracle extension interface to read and write data. JDBC materializes Oracle objects as instances of particular Java classes. Two main steps in using JDBC to access Oracle objects are:

1. Creating the Java classes for the Oracle objects
2. Populating these classes. You have the following options:
   - Let JDBC materialize the object as a `STRUCT` object.
   - Explicitly specify the mappings between Oracle objects and Java classes.

     This includes customizing your Java classes for object data. The driver then must be able to populate instances of the custom object classes that you specify. This imposes a set of constraints on the Java classes. To satisfy these constraints, you can define your classes to implement either the JDBC standard `java.sql.SQLData` interface or the Oracle extension `oracle.jdbc.OracleData` interface.

You can use the Oracle JPublisher utility to generate custom Java classes.

> **Note:** When you use the `SQLData` interface, you must use a Java type map to specify your SQL-Java mapping, unless weakly typed `java.sql.Struct` objects will suffice.

Using the Default STRUCT Class for Oracle Objects

If you choose not to supply a custom Java class for your SQL-Java mapping for an Oracle object, then Oracle JDBC materializes the object as an object that implements the `java.sql.Struct` interface.

You would typically want to use `STRUCT` objects, instead of custom Java objects, in situations where you do not know the actual SQL type. For example, your Java application might be a tool to manipulate arbitrary object data within the database, as opposed to being an end-user application. You can select data from the database into `STRUCT` objects and create `STRUCT` objects for inserting data into the database. `STRUCT` objects completely preserve data, because they maintain the data in SQL format. Using `STRUCT` objects is more efficient and more precise in situations where you do not need the information in an application specific form.

This section covers the following topics:

- Retrieving STRUCT Objects and Attributes
- Creating STRUCT Objects
- Binding STRUCT Objects into Statements
- STRUCT Automatic Attribute Buffering

Retrieving STRUCT Objects and Attributes

This section discusses how to retrieve and manipulate Oracle objects and their attributes, using either Oracle-specific features or JDBC 2.0 standard features.

> **Note:** The JDBC driver seamlessly handles embedded objects, that is, `STRUCT` objects that are attributes of `STRUCT` objects, in the same way that it typically handles objects. When the JDBC driver retrieves an attribute that is an object, it follows the same rules of conversion by using the type map, if it is available, or by using default mapping.

Retrieving an Oracle Object as a java.sql.Struct Object

Alternatively, in the preceding example, you can use standard JDBC functionality, such as `getObject`, to retrieve an Oracle object from the database as an instance of `java.sql.Struct`. The `getObject` method returns a `java.lang.Object`, so, you must cast the output of the method to `Struct`. For example:

```
ResultSet rs= stmt.executeQuery("SELECT * FROM struct_table");
java.sql.Struct jdbcStruct = (java.sql.Struct)rs.getObject(1);
```

Retrieving Attributes as oracle.sql Types

If you want to retrieve Oracle object attributes from a `STRUCT` or `Struct` instance as `oracle.sql` types, then use the `getOracleAttributes` method of the `oracle.sql.STRUCT` class, as follows:

```
oracle.sql.Datum[] attrs = oracleSTRUCT.getOracleAttributes();
```

or:

```
oracle.sql.Datum[] attrs = ((oracle.sql.STRUCT)jdbcStruct).getOracleAttributes();
```

Retrieving Attributes as Standard Java Types

If you want to retrieve Oracle object attributes as standard Java types from a `STRUCT` or `Struct` instance, use the standard `getAttributes` method:

```
Object[] attrs = jdbcStruct.getAttributes();
```

> **Note:** Oracle JDBC drivers cache array and structure descriptors. This provides enormous performance benefits. However, it means that if you change the underlying type definition of a structure type in the database, the cached descriptor for that structure type will become stale and your application will receive a `SQLException` exception.

Creating STRUCT Objects

For information about creating STRUCT objects, refer to "Overview of Class oracle.sql.STRUCT" on page 4-6.

Binding STRUCT Objects into Statements

To bind an `oracle.sql.STRUCT` object to a prepared statement or callable statement, you can either use the standard `setObject` method (specifying the type code), or cast the statement object to an Oracle statement type and use the Oracle extension `setOracleObject` method. For example:

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
Struct mySTRUCT = conn.createStruct (...);
ps.setObject(1, mySTRUCT, Types.STRUCT);
```

or:

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
Struct mySTRUCT = conn.createStruct (...);
((OraclePreparedStatement)ps).setOracleObject(1, mySTRUCT);
```

STRUCT Automatic Attribute Buffering

Oracle JDBC driver furnishes public methods to enable and disable buffering of `STRUCT` attributes.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.STRUCT` class is deprecated and replaced with the `oracle.jdbc.OracleStruct` interface, which is a part of the `oracle.jdbc` package. Oracle strongly recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleStruct` interface.

See Also: "ARRAY Automatic Element Buffering" on page 16-4

The following methods are included with the `oracle.sql.STRUCT` class:

- `public void setAutoBuffering(boolean enable)`
- `public boolean getAutoBuffering()`

The `setAutoBuffering(boolean)` method enables or disables auto-buffering. The `getAutoBuffering` method returns the current auto-buffering mode. By default, auto-buffering is disabled.

It is advisable to enable auto-buffering in a JDBC application when the `STRUCT` attributes are accessed more than once by the `getAttributes` and `getArray` methods, presuming the `ARRAY` data is able to fit into the Java Virtual Machine (JVM) memory without overflow.

Note: Buffering the converted attributes may cause the JDBC application to consume a significant amount of memory.

When you enable auto-buffering, the `oracle.sql.STRUCT` object keeps a local copy of all the converted attributes. This data is retained so that subsequent access of this information does not require going through the data format conversion process.

Creating and Using Custom Object Classes for Oracle Objects

If you want to create custom object classes for your Oracle objects, then you must define entries in the type map that specify the custom object classes that the drivers instantiate for the corresponding Oracle objects.

You must also provide a way to create and populate instances of the custom object class from the Oracle object and its attribute data. The driver must be able to read from a custom object class and write to it. In addition, the custom object class can provide `get`*`XXX`* and `set`*`XXX`* methods corresponding to the attributes of the Oracle object, although this is not necessary. To create and populate the custom classes and provide these read/write capabilities, you can choose between the following interfaces:

- The JDBC standard `SQLData` interface
- The `OracleData` and `OracleDataFactory` interfaces provided by Oracle

The custom object class you create must implement one of these interfaces. The `OracleData` interface can also be used to implement the custom reference class corresponding to the custom object class. However, if you are using the `SQLData` interface, then you can use only weak reference types in Java, such as `java.sql.Ref` or `oracle.sql.REF`. The `SQLData` interface is for mapping SQL objects only.

As an example, assume you have an Oracle object type, `EMPLOYEE`, in the database that consists of two attributes: `Name`, which is of the `CHAR` type and `EmpNum`, which is of the `NUMBER` type. You use the type map to specify that the `EMPLOYEE` object should map to a custom object class that you call `JEmployee`. You can implement either the `SQLData` or `OracleData` interface in the `JEmployee` class.

You can create custom object classes yourself, but the most convenient way to create them is to use the Oracle JPublisher utility to create them for you. JPublisher supports the standard `SQLData` interface as well as the Oracle-specific `OracleData` interface, and is able to generate classes that implement either one.

> **See Also:** "Using JPublisher to Create Custom Object Classes" on page 13-26 and "Object-Type Inheritance" on page 13-15

This section covers the following topics:

Relative Advantages of OracleData versus SQLData

In deciding which of the two interface implementations to use, you need to consider the advantages of `OracleData` and `SQLData`.

The `SQLData` interface is for mapping SQL objects only. The `OracleData` interface is more flexible, enabling you to map SQL objects as well as any other SQL type for which you want to customize processing. You can create an `OracleData`

implementation from any data type found in Oracle Database. This could be useful, for example, for serializing `RAW` data in Java.

Advantages of the OracleData Interface

The advantages of the `OracleData` interface are:

- It does not require an entry in the type map for the Oracle object.
- It has awareness of Oracle extensions.
- You can construct an `OracleData` from an `oracle.sql.STRUCT`. This is more efficient because it avoids unnecessary conversions to native Java types.
- You can obtain the corresponding JDBC object from `OracleData`, using the `toJDBCObject` method.

Advantages of SQLData

`SQLData` is a JDBC standard that makes your code portable.

Understanding Type Maps for SQLData Implementations

If you use the `SQLData` interface in a custom object class, then you must create type map entries that specify the custom object class to use in mapping the Oracle object type to Java. You can either use the default type map of the connection object or a type map that you specify when you retrieve the data from the result set. The `getObject` method of the `ResultSet` interface has a signature that lets you specify a type map. You can use either of the following:

```
rs.getObject(int columnIndex);

rs.getObject(int columnIndex, Map map);
```

> **See Also:** "Creating and Using Custom Object Classes for Oracle Objects" on page 13-5

When using a SQLData implementation, if you do not include a type map entry, then the object maps to the `oracle.jdbc.OracleStruct` interface by default. `OracleData` implementations, by contrast, have their own mapping functionality so that a type map entry is not required. When using an `OracleData` implementation, use the Oracle `getObject(int columnindex, OracleDataFactory factory)` method.

The type map relates a Java class to the SQL type name of an Oracle object. This one-to-one mapping is stored in a hash table as a keyword-value pair. When you read data from an Oracle object, the JDBC driver considers the type map to determine which Java class to use to materialize the data from the Oracle object type. When you write data to an Oracle object, the JDBC driver gets the SQL type name from the Java class by calling the `getSQLTypeName` method of the `SQLData` interface. The actual conversion between SQL and Java is performed by the driver.

The attributes of the Java class that corresponds to an Oracle object can use either Java native types or Oracle native types to store attributes.

Creating Type Map and Defining Mappings for a SQLData Implementation

When using a `SQLData` implementation, the JDBC applications programmer is responsible for providing a type map, which must be an instance of a class that implements the standard `java.util.Map` interface.

You have the option of creating your own class to accomplish this, but the standard `java.util.Hashtable` class meets the requirement.

`Hashtable` and other classes used for type maps implement a `put` method that takes keyword-value pairs as input, where each key is a fully qualified SQL type name and the corresponding value is an instance of a specified Java class.

A type map is associated with a connection instance. The standard `java.sql.Connection` interface and the Oracle-specific `oracle.jdbc.OracleConnection` interface include a `getTypeMap` method. Both return a `Map` object.

This section covers the following topics:

- Adding Entries to an Existing Type Map
- Creating a New Type Map
- Materializing Object Types not Specified in the Type Map

Adding Entries to an Existing Type Map

When a connection instance is first established, the default type map is empty. You must populate it.

Perform the following general steps to add entries to an existing type map:

1. Use the `getTypeMap` method of your `OracleConnection` object to return the type map object of the connection. The `getTypeMap` method returns a `java.util.Map` object. For example, presuming an `OracleConnection` instance `oraconn`:

   ```
   java.util.Map myMap = oraconn.getTypeMap();
   ```

   Note: If the type map in the `OracleConnection` instance has not been initialized, then the first call to `getTypeMap` returns an empty map.

2. Use the `put` method of the type map to add map entries. The `put` method takes two arguments: a SQL type name string and an instance of a specified Java class that you want to map to.

   ```
   myMap.put(sqlTypeName, classObject);
   ```

   The *`sqlTypeName`* is a string that represents the fully qualified name of the SQL type in the database. The *`classObject`* is the Java class object to which you want to map the SQL type. Get the class object with the `Class.forName` method, as follows:

   ```
   myMap.put(sqlTypeName, Class.forName(className));
   ```

   For example, if you have a `PERSON` SQL data type defined in the `CORPORATE` database schema, then map it to a `Person` Java class defined as `Person` with this statement:

   ```
   myMap.put("CORPORATE.PERSON", Class.forName("Person"));
   oraconn.setTypeMap(newMap);
   ```

   The map has an entry that maps the `PERSON` SQL data type in the `CORPORATE` database to the `Person` Java class.

Note: SQL type names in the type map must be all uppercase, because that is how Oracle Database stores SQL names.

Creating a New Type Map

Perform the following general steps to create a new type map. This example uses an instance of `java.util.Hashtable`, which extends `java.util.Dictionary` and implements `java.util.Map`.

1. Create a new type map object.

   ```
   Hashtable newMap = new Hashtable();
   ```

2. Use the `put` method of the type map object to add entries to the map. For example, if you have an `EMPLOYEE` SQL type defined in the `CORPORATE` database, then you can map it to an `Employee` class object defined by `Employee.java`, as follows:

   ```
   newMap.put("CORPORATE.EMPLOYEE", class.forName("Employee"));
   ```

3. When you finish adding entries to the map, you must use the `setTypeMap` method of the `OracleConnection` object to overwrite the existing type map of the connection. For example:

   ```
   oraconn.setTypeMap(newMap);
   ```

   In this example, the `setTypeMap` method overwrites the original map of the `oraconn` connection object with `newMap`.

Note: The default type map of a connection instance is used when mapping is required but no map name is specified, such as for a result set `getObject` call that does not specify the map as input.

Materializing Object Types not Specified in the Type Map

If you do not provide a type map with an appropriate entry when using a `getObject` call, then the JDBC driver will materialize an Oracle object as an instance of the `oracle.jdbc.OracleStruct` interface. If the Oracle object type contains embedded objects and they are not present in the type map, then the driver will materialize the embedded objects as instances of `oracle.jdbc.OracleStruct` as well. If the embedded objects are present in the type map, then a call to the `getAttributes` method will return embedded objects as instances of the specified Java classes from the type map.

Reading and Writing Data with a SQLData Implementation

This section describes how to read data from an Oracle object or write data to an Oracle object if your corresponding Java class implements `SQLData`.

Reading SQLData Objects from a Result Set

The following text summarizes the steps to read data from an Oracle object into your Java application when you choose the `SQLData` implementation for your custom object class.

These steps assume you have already defined the Oracle object type, created the corresponding custom object class, updated the type map to define the mapping between the Oracle object and the Java class, and defined a statement object `stmt`.

1. Query the database to read the Oracle object into a JDBC result set.

```
ResultSet rs = stmt.executeQuery("SELECT emp_col FROM personnel");
```

The PERSONNEL table contains one column, EMP_COL, of SQL type EMP_OBJECT. This SQL type is defined in the type map to map to the Java class Employee.

2. Use the getObject method of Oracle result set to populate an instance of your custom object class with data from one row of the result set. The getObject method returns the user-defined SQLData object because the type map contains an entry for Employee.

```
if (rs.next())
   Employee emp = (Employee)rs.getObject(1);
```

Note that if the type map did not have an entry for the object, then the getObject method will return an oracle.jdbc.OracleStruct object. Cast the output to type OracleStruct because the getObject method signature returns the generic java.lang.Object type.

```
if (rs.next())
   OracleStruct empstruct = (OracleStruct)rs.getObject(1);
```

The getObject method calls readSQL, which, in turn, calls read*XXX* from the SQLData interface.

Note: If you want to avoid using the defined type map, then use the getSTRUCT method. This method always returns a STRUCT object, even if there is a mapping entry in the type map.

---

3. If you have get methods in your custom object class, then use them to read data from your object attributes. For example, if EMPLOYEE has the attributes EmpName of type CHAR and EmpNum of type NUMBER, then provide a getEmpName method that returns a Java String and a getEmpNum method that returns an int value. Then call them in your Java application, as follows:

```
String empname = emp.getEmpName();
int empnumber = emp.getEmpNum();
```

Retrieving SQLData Objects from a Callable Statement OUT Parameter

Consider you have a CallableStatement instance, cs, that calls a PL/SQL function GETEMPLOYEE. The program passes an employee number to the function. The function returns the corresponding Employee object. To retrieve this object you do the following:

1. Prepare a CallableStatement to call the GETEMPLOYEE function, as follows:

```
CallableStatement ocs = conn.prepareCall("{ ? = call GETEMPLOYEE(?) }");
```

2. Declare the empnumber as the input parameter to GETEMPLOYEE. Register the SQLData object as the OUT parameter, with the type code OracleTypes.STRUCT. Then, run the statement. This can be done as follows:

```
cs.setInt(2, empnumber);
cs.registerOutParameter(1, OracleTypes.STRUCT, "EMP_OBJECT");
cs.execute();
```

3. Use the getObject method to retrieve the employee object.

```
Employee emp = (Employee)cs.getObject(1);
```

If there is no type map entry, then the `getObject` method will return a `java.sql.Struct` object.

```
Struct emp = cs.getObject(1);
```

Passing SQLData Objects to a Callable Statement as an IN Parameter

Suppose you have a PL/SQL function `addEmployee(?)` that takes an `Employee` object as an `IN` parameter and adds it to the `PERSONNEL` table. In this example, `emp` is a valid `Employee` object.

1. Prepare an `CallableStatement` to call the `addEmployee(?)` function.

```
CallableStatement cs =
  conn.prepareCall("{ call addEmployee(?) }");
```

2. Use `setObject` to pass the `emp` object as an `IN` parameter to the callable statement. Then, call the statement.

```
cs.setObject(1, emp);
cs.execute();
```

Writing Data to an Oracle Object Using a SQLData Implementation

The following text describes the steps in writing data to an Oracle object from your Java application when you choose the `SQLData` implementation for your custom object class.

This description assumes you have already defined the Oracle object type, created the corresponding Java class, and updated the type map to define the mapping between the Oracle object and the Java class.

1. If you have `set` methods in your custom object class, then use them to write data from Java variables in your application to attributes of your Java data type object.

```
emp.setEmpName(empname);
emp.setEmpNum(empnumber);
```

2. Prepare a statement that updates an Oracle object in a row of a database table, as appropriate, using the data provided in your Java data type object.

```
PreparedStatement pstmt = conn.prepareStatement
                          ("INSERT INTO PERSONNEL VALUES (?)");
```

3. Use the `setObject` method of the prepared statement to bind your Java data type object to the prepared statement.

```
pstmt.setObject(1, emp);
```

4. Run the statement, which updates the database.

```
pstmt.executeUpdate();
```

Understanding the OracleData Interface

You can create a custom object class that implements the `oracle.jdbc.OracleData` and the `oracle.jdbc.OracleDataFactory` interfaces to make an Oracle object and its attribute data available to Java applications. The `OracleData` and `OracleDataFactory` interfaces are Oracle-specific and are not a part of the JDBC standard.

Note:

- The JPublisher utility supports the generation of classes that implement the `OracleData` and `OracleDataFactory` interfaces.
- Starting from Oracle Database 12*c* Release 1 (12.1), the `OracleData` and the `OracleDataFactory` interfaces replace the `ORAData` and the `ORADataFactory` interfaces.

Understanding the OracleData Interface Features

The `OracleData` interface has the following advantages:

- It supports Oracle extensions to the standard JDBC types.
- It does not require a type map to specify the names of the Java custom classes you want to create.
- It provides better performance. `OracleData` works directly with `Datum` types, the internal format the driver uses to hold Oracle objects.

The `OracleData` and the `OracleDataFactory` interfaces perform the following:

- The `toJDBCObject` method of the `OracleData` class transforms the data into an `oracle.jdbc.*` representation.
- `OracleDataFactory` specifies a `create` method equivalent to a constructor for the custom object class. It creates and returns an `OracleData` instance. The JDBC driver uses the `create` method to return an instance of the custom object class to your Java application or applet. It takes as input a `java.lang.Object` object and an integer indicating the corresponding SQL type code as specified in the `OracleTypes` class.

`OracleData` and `OracleDataFactory` have the following definitions:

```
package oracle.jdbc;
import java.sql.Connection;
import java.sql.SQLException;
public interface OracleData
{
   public Object toJDBCObject(Connection conn) throws SQLException;
}

package oracle.jdbc;
import java.sql.SQLException;
public interface OracleDataFactory
{
    public OracleData create(Object jdbcValue, int sqlType) throws SQLException;

}
```

Where *conn* represents the Connection object, *jdbcValue* represents an object of type `java.lang.object` that is to be used to initialize the Object being created, and *sqlType* represents the SQL type of the specified `Datum` object.

Retrieving and Inserting Object Data

The JDBC drivers provide the following methods to retrieve and insert object data as instances of `OracleData`.

You can retrieve the object data in one of the following ways:

- Use the following `getObject` method of the Oracle-specific `OracleResultSet` interface:

  ```
  ors.getObject(int col_index, OracleDataFactory factory
  );
  ```

  This method takes as input the column index of the data in your result set and an `OracleDataFactory` instance. For example, you can implement a `getOracleDataFactory` method in your custom object class to produce the `OracleDataFactory` instance to input to the `getObject` method. The type map is not required when using Java classes that implement `OracleData`.

- Use the standard `getObject(`*`index`*`, `*`map`*`)` method specified by the `ResultSet` interface to retrieve data as instances of `OracleData`. In this case, you must have an entry in the type map that identifies the factory class to be used for the given object type and its corresponding SQL type name.

You can insert object data in one of the following ways:

- Use the following `setObject` method of the Oracle-specific `OraclePreparedStatement` class:

  ```
  setObject(int bind_index, Object custom_object);
  ```

  This method takes as input the parameter index of the bind variable and an instance of `OracleData` as the name of the object containing the variable.

- Use the standard `setObject` method specified by the `PreparedStatement` interface. You can also use this method, in its different forms, to insert `OracleData` instances without requiring a type map.

The following sections describe the `getObject` and `setObject` methods.

To continue the example of an Oracle object `EMPLOYEE`, you might have something like the following in your Java application:

```
OracleData obj = ors.getObject(1, Employee.getOracleDataFactory());
```

In this example, `ors` is an instance of the `OracleResultSet` interface, `getObject` is a method in the `OracleResultSet` interface used to retrieve an `OracleData` object, and the `EMPLOYEE` is in column 1 of the result set. The `static Employee.getOracleDataFactory` method will return an `OracleDataFactory` to the JDBC driver. The JDBC driver will call `create()` from this object, returning to your Java application an instance of the `Employee` class populated with data from the result set.

Note:

- `OracleData` and `OracleDataFactory` are defined as separate interfaces so that different Java classes can implement them if you wish.
- To use the `OracleData` interface, your custom object classes must import `oracle.jdbc.*`.

Reading and Writing Data with an OracleData Implementation

This section describes how to read data from an Oracle object or write data to an Oracle object if your corresponding Java class implements `OracleData`.

Reading Data from an Oracle Object Using an OracleData Implementation

The following text summarizes the steps in reading data from an Oracle object into your Java application. These steps apply whether you implement `OracleData` manually or use JPublisher to produce your custom object classes.

These steps assume you have already defined the Oracle object type, created the corresponding custom object class or had JPublisher create it for you, and defined a statement object `stmt`.

1. Query the database to read the Oracle object into a result set, casting it to an Oracle result set.

   ```
   OracleResultSet ors = (OracleResultSet)stmt.executeQuery
                         ("SELECT Emp_col FROM PERSONNEL");
   ```

   Where `PERSONNEL` is a one-column table. The column name is `Emp_col` of type `Employee_object`.

2. Use the `getObject` method of Oracle result set to populate an instance of your custom object class with data from one row of the result set. The `getObject` method returns a `java.lang.Object` object, which you can cast to your specific custom object class.

   ```
   if (ors.next())
      Employee emp = (Employee)ors.getObject(1, Employee.getOracleDataFactory());
   ```

   or:

   ```
   if (ors.next())
      Object obj = ors.getObject(1, Employee.getOracleDataFactory());
   ```

   This example assumes that `Employee` is the name of your custom object class and `ors` is the name of your `OracleResultSet` instance.

   For example, if the SQL type name for your object is `EMPLOYEE`, then the corresponding Java class is `Employee`, which will implement `OracleData`. The corresponding Factory class is `EmployeeFactory`, which will implement `OracleDataFactory`.

   Use this statement to declare the `EmployeeFactory` entry for your type map:

   ```
   map.put ("EMPLOYEE", Class.forName ("EmployeeFactory"));
   ```

   Then use the form of `getObject` where you specify the map object:

   ```
   Employee emp = (Employee) rs.getObject (1, map);
   ```

   If the default type map of the connection already has an entry that identifies the factory class to be used for the given object type and its corresponding SQL type name, then you can use this form of `getObject`:

   ```
   Employee emp = (Employee) rs.getObject (1);
   ```

3. If you have `get` methods in your custom object class, then use them to read data from your object attributes into Java variables in your application. For example, if `EMPLOYEE` has `EmpName` of type `CHAR` and `EmpNum` of type `NUMBER`, provide a `getEmpName` method that returns a Java `String` and a `getEmpNum` method that returns an integer. Then call them in your Java application as follows:

   ```
   String empname = emp.getEmpName();
   int empnumber = emp.getEmpNum();
   ```

Writing Data to an Oracle Object Using an OracleData Implementation

The following text summarizes the steps in writing data to an Oracle object from your Java application. These steps apply whether you implement `OracleData` manually or use JPublisher to produce your custom object classes.

These steps assume you have already defined the Oracle object type and created the corresponding custom object class.

> **Note:** The type map is not used when you are performing database `INSERT` and `UPDATE` operations.

1. If you have `set` methods in your custom object class, then use them to write data from Java variables in your application to attributes of your Java data type object.

   ```
   emp.setEmpName(empname);
   emp.setEmpNum(empnumber);
   ```

2. Write an Oracle prepared statement that updates an Oracle object in a row of a database table, as appropriate, using the data provided in your Java data type object.

   ```
   OraclePreparedStatement opstmt = conn.prepareStatement
      ("UPDATE PERSONNEL SET Employee = ? WHERE Employee.EmpNum = 28959);
   ```

   This assumes `conn` is your `Connection` object.

3. Use the `setObject` method of the `OraclePreparedStatement` interface to bind your Java data type object to the prepared statement.

   ```
   opstmt.setObject(1,emp);
   ```

   The `setObject` method calls the `toJDBCObject` method of the custom object class instance to retrieve an `oracle.jdbc.OracleStruct` object that can be written to the database.

> **Note:** You can use your Java data type objects as either `IN` or `OUT` bind variables.

Additional Uses for OracleData

The `OracleData` interface offers far more flexibility than the `SQLData` interface. The `SQLData` interface is designed to let you customize the mapping of only Oracle object types to Java types of your choice. Implementing the `SQLData` interface lets the JDBC driver populate fields of a custom Java class instance from the original SQL object data, and the reverse, after performing the appropriate conversions between Java and SQL types.

The `OracleData` interface goes beyond supporting the customization of Oracle object types to Java types. It lets you provide a mapping between Java object types and *any* SQL type supported by the `oracle.sql` package.

You may find it useful to provide custom Java classes to wrap `oracle.sql.*` types and then implement customized conversions or functionality as well. The following are some possible scenarios:

- Performing encryption and decryption or validation of data
- Performing logging of values that have been read or are being written

- Parsing character columns, such as character fields containing URL information, into smaller components
- Mapping character strings into numeric constants
- Making data into more desirable Java formats, such as mapping a `DATE` field to `java.util.Date` format
- Customizing data representation, for example, data in a table column is in feet but you want it represented in meters after it is selected
- Serializing and deserializing Java objects

For example, use `OracleData` to store instances of Java objects that do not correspond to a particular SQL object type in the database in columns of SQL type `RAW`. The `create` method in `OracleDataFactory` would have to implement a conversion from an object of type `oracle.sql.RAW` to the desired Java object. The `toJDBCObject` method in `OracleData` would have to implement a conversion from the Java object to an `oracle.sql.RAW` object. You can also achieve this using Java serialization.

Upon retrieval, the JDBC driver transparently retrieves the raw bytes of data in the form of an `oracle.sql.RAW` and calls the `create` method of `OracleDataFactory` to convert the `oracle.sql.RAW` object to the desired Java class.

When you insert the Java object into the database, you can simply bind it to a column of type `RAW` to store it. The driver transparently calls the `OracleData.toJDBCObject` method to convert the Java object to an `oracle.sql.RAW` object. This object is then stored in a column of type `RAW` in the database.

Support for the `OracleData` interfaces is also highly efficient because the conversions are designed to work using `oracle.sql.*` formats, which happen to be the internal formats used by the JDBC drivers. Moreover, the type map, which is necessary for the `SQLData` interface, is not required when using Java classes that implement `OracleData`.

See Also: "Understanding the OracleData Interface" on page 13-10

Object-Type Inheritance

Object-type inheritance allows a new object type to be created by extending another object type. The new object type is then a subtype of the object type from which it extends. The subtype automatically inherits all the attributes and methods defined in the supertype. The subtype can add attributes and methods and overload or override methods inherited from the supertype.

Object-type inheritance introduces **substitutability**. Substitutability is the ability of a slot declared to hold a value of type `T` in addition to any subtype of type `T`. Oracle JDBC drivers handle substitutability transparently.

A database object is returned with its most specific type without losing information. For example, if the `STUDENT_T` object is stored in a `PERSON_T` slot, Oracle JDBC driver returns a Java object that represents the `STUDENT_T` object.

This section covers the following topics:

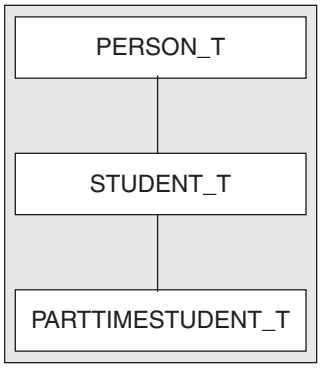
- Accessing Subtype Data Fields
- Inheritance Metadata Methods

Creating Subtypes

Create custom object classes if you want to have Java classes that explicitly correspond to the Oracle object types. If you have a hierarchy of object types, you may want a corresponding hierarchy of Java classes.

The most common way to create a database subtype in JDBC is to run a SQL `CREATE TYPE` command using the `execute` method of the `java.sql.Statement` interface. For example, you want to create a type inheritance hierarchy as depicted in Figure 13–1:

Figure 13–1 Type Inheritance Hierarchy



The JDBC code for this can be as follows:

```
Statement s = conn.createStatement();
s.execute ("CREATE TYPE Person_T (SSN NUMBER, name VARCHAR2(30),
  address VARCHAR2(255))");
s.execute ("CREATE TYPE Student_T UNDER Person_t (deptid NUMBER,
  major VARCHAR2(100))");
s.execute ("CREATE TYPE PartTimeStudent_t UNDER Student_t (numHours NUMBER)");
```

In the following code, the `foo` member procedure in type `ST` is overloaded and the member procedure `print` overwrites the copy it inherits from type `T`.

```
CREATE TYPE T AS OBJECT (...,
  MEMBER PROCEDURE foo(x NUMBER),
  MEMBER PROCEDURE Print(),
  ...
  NOT FINAL;

CREATE TYPE ST UNDER T (...,
  MEMBER PROCEDURE foo(x DATE),         <-- overload "foo"
  OVERRIDING MEMBER PROCEDURE Print(),   <-- override "print"
  STATIC FUNCTION bar(...) ...
  ...
  );
```

Once the subtypes have been created, they can be used as both columns of a base table as well as attributes of a object type.

See Also: *Oracle Database Object-Relational Developer's Guide*

Implementing Customized Classes for Subtypes

In most cases, a customized Java class represents a database object type. When you create a customized Java class for a subtype, the Java class can either mirror the database object type hierarchy or not.

You can use either the `OracleData` or `SQLData` solution in creating classes to map to the hierarchy of object types.

This section covers the following topics:

- Use of OracleData for Type Inheritance Hierarchy
- Use of SQLData for Type Inheritance Hierarchy
- JPublisher Utility

Use of OracleData for Type Inheritance Hierarchy

Oracle recommends customized mappings, where Java classes implement the `oracle.sql.OracleData` interface. `OracleData` mapping requires the JDBC application to implement the `OracleData` and `OracleDataFactory` interfaces. The class implementing the `OracleDataFactory` interface contains a factory method that produces objects. Each object represents a database object.

The hierarchy of the class implementing the `OracleData` interface can mirror the database object type hierarchy. For example, the Java classes mapping to `PERSON_T` and `STUDENT_T` are as follows:

Person.java using OracleData

Code for the `Person.java` class which implements the `OracleData` and `OracleDataFactory` interfaces:

```
public static OracleDataFactory getOracleDataFactory()
  {
    return _personFactory;
  }

  public Person () {}

  public Person(NUMBER ssn, CHAR name, CHAR address)
  {
    this.ssn = ssn;
    this.name = name;
    this.address = address;
  }

  public Object toJDBCObject(OracleConnection c) throws SQLException
  {
    Object [] attributes = { ssn, name, address };
 Struct struct = c.createStruct("HR.PERSON_T", attributes);
    return struct;
  }

  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
  {
    if (d == null) return null;
    Object [] attributes = ((STRUCT) d).getOracleAttributes();
    return new Person((NUMBER) attributes[0],
                      (CHAR) attributes[1],
                      (CHAR) attributes[2]);
  }
```

```
}
```

Student.java extending Person.java

Code for the `Student.java` class, which extends the `Person.java` class:

```
class Student extends Person
{
  static final Student _studentFactory = new Student ();

  public NUMBER deptid;
  public CHAR major;

  public static OracleDataFactory getOracleDataFactory()
  {
    return _studentFactory;
  }

  public Student () {}

  public Student (NUMBER ssn, CHAR name, CHAR address,
                  NUMBER deptid, CHAR major)
  {
    super (ssn, name, address);
    this.deptid = deptid;
    this.major = major;
  }

  public Object toJDBCObject(OracleConnection c) throws SQLException
  {
    Object [] attributes = { ssn, name, address, deptid, major };
    Struct struct = c.createStruct("HR.STUDENT_T", attributes);
    return struct;
  }

  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
  {
    if (d == null) return null;
    Object [] attributes = ((STRUCT) d).getOracleAttributes();
    return new Student((NUMBER) attributes[0],
                       (CHAR) attributes[1],
                       (CHAR) attributes[2],
                       (NUMBER) attributes[3],
                       (CHAR) attributes[4]);
  }
}
```

Customized classes that implement the `OracleData` interface do not have to mirror the database object type hierarchy. For example, you could have declared the `Student` class without a superclass. In this case, `Student` would contain fields to hold the inherited attributes from `PERSON_T` as well as the attributes declared by `STUDENT_T`.

OracleDataFactory Implementation

The JDBC application uses the factory class in querying the database to return instances of `Person` or its subclasses, as in the following example:

```
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
```

```
  rset.getOracleData(1,Person.getOracleDataFactory());
  ...
}
```

A class implementing the `OracleDataFactory` interface should be able to produce instances of the associated custom object type, as well as instances of any subtype, or at least all the types you expect to support.

In the following example, the `PersonFactory.getOracleDataFactory` method returns a factory that can handle `PERSON_T`, `STUDENT_T`, and `PARTTIMESTUDENT_T` objects, by returning `person`, `student`, or `parttimestudent` Java instances.

```
class PersonFactory implements OracleDataFactory
{
  static final PersonFactory _factory = new PersonFactory ();

  public static OracleDataFactory getOracleDataFactory()
  {
    return _factory;
  }

  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
  {
    STRUCT s = (STRUCT) jdbcValue;
    if (s.getSQLTypeName ().equals ("HR.PERSON_T"))
      return Person.getOracleDataFactory ().create (jdbcValue, sqlType);
    else if (s.getSQLTypeName ().equals ("HR.STUDENT_T"))
      return Student.getOracleDataFactory ().create(jdbcValue, sqlType);
    else if (s.getSQLTypeName ().equals ("HR.PARTTIMESTUDENT_T"))
      return ParttimeStudent.getOracleDataFactory ().create(jdbcValue, sqlType);
    else
      return null;
  }
}
```

The following example assumes a table `tabl1`, such as the following:

```
CREATE TABLE tabl1 (idx NUMBER, person PERSON_T);
INSERT INTO tabl1 VALUES (1, PERSON_T (1000, 'HR', '100 Oracle Parkway'));
INSERT INTO tabl1 VALUES (2, STUDENT_T (1001, 'Peter', '200 Oracle Parkway', 101,
'CS'));
INSERT INTO tabl1 VALUES (3, PARTTIMESTUDENT_T (1002, 'David', '300 Oracle
Parkway', 102, 'EE'));
```

Use of SQLData for Type Inheritance Hierarchy

The customized classes that implement the `java.sql.SQLData` interface can mirror the database object type hierarchy. The `readSQL` and `writeSQL` methods of a subclass typically call the corresponding superclass methods to read or write the superclass attributes before reading or writing the subclass attributes. For example, the Java classes mapping to `PERSON_T` and `STUDENT_T` are as follows:

Person.java using SQLData

Code for the `Person.java` class, which implements the `SQLData` interface:

```
import java.sql.*;

public class Person implements SQLData
{
  private String sql_type;
```

```
  public int ssn;
  public String name;
  public String address;

  public Person () {}

  public String getSQLTypeName() throws SQLException { return sql_type; }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readString();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeString (address);
  }
}
```

Student.java extending Student.java

Code for the `Student.java` class, which extends the `Person.java` class:

```
import java.sql.*;

public class Student extends Person
{
  private String sql_type;
  public int deptid;
  public String major;

  public Student () { super(); }

  public String getSQLTypeName() throws SQLException { return sql_type; }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    super.readSQL (stream, typeName);    // read supertype attributes
    sql_type = typeName;
    deptid = stream.readInt();
    major = stream.readString();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    super.writeSQL (stream);        // write supertype
                                    // attributes
    stream.writeInt (deptid);
    stream.writeString (major);
  }
}
```

Although not required, it is recommended that the customized classes, which implement the `SQLData` interface, mirror the database object type hierarchy. For example, you could have declared the `Student` class without a superclass. In this case,

Student would contain fields to hold the inherited attributes from PERSON_T as well as the attributes declared by STUDENT_T.

Student.java using SQLData

Code for the Student.java class, which does not extend the Person.java class, but implements the SQLData interface directly:

```
import java.sql.*;

public class Student implements SQLData
{
  private String sql_type;

  public int ssn;
  public String name;
  public String address;
  public int deptid;
  public String major;

  public Student () {}

  public String getSQLTypeName() throws SQLException { return sql_type; }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readString();
    deptid = stream.readInt();
    major = stream.readString();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeString (address);
    stream.writeInt (deptid);
    stream.writeString (major);
  }
}
```

JPublisher Utility

Even though you can manually create customized classes that implement the SQLData, OracleData, and OracleDataFactory interfaces, it is recommended that you use Oracle JPublisher to automatically generate these classes. The customized classes generated by Oracle JPublisher that implement the SQLData, OracleData, and OracleDataFactory interfaces, can mirror the inheritance hierarchy.

See Also:

- "Using JPublisher to Create Custom Object Classes" on page 13-26
- *Oracle Database JPublisher User's Guide*

Retrieving Subtype Objects

In a typical JDBC application, a subtype object is returned as one of the following:

- A query result
- A PL/SQL `OUT` parameter
- A type attribute

You can use either the default mapping or the `SQLData` mapping or the `OracleData` mapping to retrieve a subtype.

Using Default Mapping

By default, a database object is returned as an instance of the `oracle.jdbc.OracleStruct` interface. This instance may represent an object of either the declared type or subtype of the declared type. If the `OracleStruct` interface represents a subtype object in the database, then it contains the attributes of its supertype as well as those defined in the subtype.

Oracle JDBC driver returns database objects in their most specific type. The JDBC application can use the `getSQLTypeName` method of the `OracleStruct` interface to determine the SQL type of the `STRUCT` object. The following code shows this:

```
// tab1.person column can store PERSON_T, STUDENT_T and PARTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  oracle.sql.STRUCT s = (oracle.sql.STRUCT) rset.getObject(1);
  if (s != null)
    System.out.println (s.getSQLTypeName());    // print out the type name which
    // may be HR.PERSON_T, HR.STUDENT_T or HR.PARTTIMESTUDENT_T
}
```

Using SQLData Mapping

With `SQLData` mapping, the JDBC driver returns the database object as an instance of the class implementing the `SQLData` interface.

To use `SQLData` mapping in retrieving database objects, do the following:

1. Implement the container classes that implement the `SQLData` interface for the desired object types.
2. Populate the connection type map with entries that specify what custom Java type corresponds to each Oracle object type.
3. Use the `getObject` method to access the SQL object values.

   The JDBC driver checks the type map for an entry match. If one exists, then the driver returns the database object as an instance of the class implementing the `SQLData` interface.

The following code shows the whole SQLData customized mapping process:

```
// The JDBC application developer implements Person.java for PERSON_T,
// Student.java for STUDENT_T
// and ParttimeStudent.java for PARTTIMESTUDEN_T.

Connection conn = ...;  // make a JDBC connection

// obtains the connection typemap
java.util.Map map = conn.getTypeMap ();
```

```
// populate the type map
map.put ("HR.PERSON_T", Class.forName ("Person"));
map.put ("HR.STUDENT_T", Class.forName ("Student"));
map.put ("HR.PARTTIMESTUDENT_T", Class.forName ("ParttimeStudent"));

// tab1.person column can store PERSON_T, STUDENT_T and PARTTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  // "s" is instance of Person, Student or ParttimeStudent
  Object s = rset.getObject(1);

  if (s != null)
  {
    if (s instanceof Person)
      System.out.println ("This is a Person");
    else if (s instanceof Student)
      System.out.println ("This is a Student");
    else if (s instanceof ParttimeStudent)
      System.out.pritnln ("This is a PartimeStudent");
    else
      System.out.println ("Unknown type");
  }
}
```

The JDBC drivers check the connection type map for each call to the following:

- `getObject` method of the `java.sql.ResultSet` and `java.sql.CallableStatement` interfaces
- `getAttribute` method of the `java.sql.Struct` interface
- `getArray` method of the `java.sql.Array` interface
- `getValue` method of the `oracle.sql.REF` interface

Using OracleData Mapping

With `OracleData` mapping, the JDBC driver returns the database object as an instance of the class implementing the `OracleData` interface.

Oracle JDBC driver needs to be informed of what Java class is mapped to the Oracle object type. The following are the two ways to inform Oracle JDBC drivers:

- The JDBC application uses the `getObject(int idx, OracleDataFactory f)` method to access database objects. The second parameter of the `getObject` method specifies an instance of the factory class that produces the customized class. The `getObject` method is available in the `OracleResultSet` and `OracleCallableStatement` interfaces.
- The JDBC application populates the connection type map with entries that specify what custom Java type corresponds to each Oracle object type. The `getObject` method is used to access the Oracle object values.

The second approach involves the use of the standard `getObject` method. The following code example demonstrates the first approach:

```
// tab1.person column can store both PERSON_T and STUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  Object s = rset.getObject(1, PersonFactory.getOracleDataFactory());
```

```
  if (s != null)
  {
    if (s instanceof Person)
      System.out.println ("This is a Person");
    else if (s instanceof Student)
      System.out.println ("This is a Student");
    else if (s instanceof ParttimeStudent)
      System.out.pritnln ("This is a PartimeStudent");
    else
      System.out.println ("Unknown type");
  }
}
```

Creating Subtype Objects

There are cases where JDBC applications create database subtype objects with JDBC drivers. These objects are sent either to the database as bind variables or are used to exchange information within the JDBC application.

With customized mapping, the JDBC application creates either `SQLData`-based or `OracleData`-based objects, depending on the approach you choose, to represent database subtype objects. With default mapping, the JDBC application creates `STRUCT` objects to represent database subtype objects. All the data fields inherited from the supertype as well as all the fields defined in the subtype must have values. The following code demonstrates this:

```
Connection conn = ...   // make a JDBC connection
...
Object[] attrs = {
  new Integer(1234), "HR", "500 Oracle Parkway", // data fields defined in
                                                 // PERSON_T
  new Integer(102), "CS",                        // data fields defined in
                                                 // STUDENT_T
  new Integer(4)                                 // data fields defined in
                                                 // PARTTIMESTUDENT_T
};
Struct s = conn.createStruct("HR.PARTTIMESTUDENT", attrs);
```

`s` is initialized with data fields inherited from `PERSON_T` and `STUDENT_T`, and data fields defined in `PARTTIMESTUDENT_T`.

Sending Subtype Objects

In a typical JDBC application, a Java object that represents a database object is sent to the databases as one of the following:

- A data manipulation language (DML) bind variable
- A PL/SQL `IN` parameter
- An object type attribute value

The Java object can be an instance of the `STRUCT` class or an instance of the class implementing either the `SQLData` or `OracleData` interface. Oracle JDBC driver will convert the Java object into the linearized format acceptable to the database SQL engine. Binding a subtype object is the same as binding a standard object.

Accessing Subtype Data Fields

While the logic to access subtype data fields is part of the customized class, this logic for default mapping is defined in the JDBC application itself. The database objects are returned as instances of the `oracle.jdbc.OracleStruct` class. The JDBC application needs to call one of the following access methods in the `STRUCT` class to access the data fields:

- `Object[] getAttribute()`
- `oracle.sql.Datum[] getOracleAttribute()`

Subtype Data Fields from the getAttribute Method

The `getAttribute` method of the `java.sql.Struct` interface is used in JDBC 2.0 to access object data fields. This method returns a `java.lang.Object` array, where each array element represents an object attribute. You can determine the individual element type by referencing the corresponding attribute type in the JDBC conversion matrix. For example, a SQL `NUMBER` attribute is converted to a `java.math.BigDecimal` object. The `getAttribute` method returns all the data fields defined in the supertype of the object type as well as data fields defined in the subtype. The supertype data fields are listed first followed by the subtype data fields.

Subtype Data Fields from the getOracleAttribute Method

The `getOracleAttribute` method is an Oracle extension method and is more efficient than the `getAttribute` method. The `getOracleAttribute` method returns an `oracle.sql.Datum` array to hold the data fields. Each element in the `oracle.sql.Datum` array represents an attribute. You can determine the individual element type by referencing the corresponding attribute type in the Oracle conversion matrix. For example, a SQL `NUMBER` attribute is converted to an `oracle.sql.NUMBER` object. The `getOracleAttribute` method returns all the attributes defined in the supertype of the object type, as well as attributes defined in the subtype. The supertype data fields are listed first followed by the subtype data fields.

The following code shows the use of the `getAttribute` method:

```
// tab1.person column can store PERSON_T, STUDENT_T and PARTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  oracle.sql.STRUCT s = (oracle.sql.STRUCT) rset.getObject(1);
  if (s != null)
  {
    String sqlname = s.getSQLTypeName();

    Object[] attrs = s.getAttribute();

    if (sqlname.equals ("HR.PERSON")
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
    }
    else if (sqlname.equals ("HR.STUDENT"))
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
      System.out.println ("deptid="+((BigDecimal)attrs[3]).intValue());
      System.out.println ("major="+((String)attrs[4]));
```

```
    }
    else if (sqlname.equals ("HR.PARTTIMESTUDENT"))
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
      System.out.println ("deptid="+((BigDecimal)attrs[3]).intValue());
      System.out.println ("major="+((String)attrs[4]));
      System.out.println ("numHours="+((BigDecimal)attrs[5]).intValue());
    }
    else
      throw new Exception ("Invalid type name: "+sqlname);
  }
}
rset.close ();
stmt.close ();
conn.close ();
```

Inheritance Metadata Methods

Oracle JDBC drivers provide a set of metadata methods to access inheritance properties. The inheritance metadata methods are defined in the `oracle.sql.StructDescriptor` and `oracle.jdbc.StructMetaData` classes.

The `StructMetaData` class provides inheritance metadata methods for subtype attributes. The `getMetaData` method of the `StructDescriptor` class returns an instance of `StructMetaData` of the type. The `StructMetaData` class contains the following inheritance metadata methods:

Using JPublisher to Create Custom Object Classes

A convenient way to create custom object classes, as well as other kinds of custom Java classes, is to use the Oracle JPublisher utility. It generates a full definition for a custom Java class, which you can instantiate to hold the data from an Oracle object. JPublisher-generated classes include methods to convert data from SQL to Java and from Java to SQL, as well as getter and setter methods for the object attributes.

This section covers the following topics:

- JPublisher Functionality
- JPublisher Type Mappings

See Also: *Oracle Database JPublisher User's Guide.*

JPublisher Functionality

You can direct JPublisher to create custom object classes that implement either the `SQLData` interface or the `OracleData` interface, according to how you set the JPublisher type mappings.

If you use the `OracleData` interface, then JPublisher will also create a custom reference class to map to object references for the Oracle object type. If you use the `SQLData` interface, then JPublisher will not produce a custom reference class. You would use standard `java.sql.Ref` instances instead.

If you want additional functionality, you can subclass the custom object class and add features as desired. When you run JPublisher, there is a command-line option for specifying both a generated class name and the name of the subclass you will

implement. For the SQL-Java mapping to work properly, JPublisher must know the subclass name, which is incorporated into some of the functionality of the generated class.

> **Note:** Hand-editing the JPublisher-generated class, instead of subclassing it, is not recommended. If you hand-edit this class and later have to re-run JPublisher for some reason, you would have to re-implement your changes.

JPublisher Type Mappings

JPublisher offers various choices for how to map user-defined types and their attribute types between SQL and Java. This section lists categories of SQL types and the mapping options available for each category.

Categories of SQL Types

JPublisher categorizes SQL types into the following groups, with corresponding JPublisher options as specifies:

- User-defined types (UDT)

  This includes Oracle objects, references, and collections. You use the JPublisher `-usertypes` option to specify the type-mapping implementation for UDTs, either a standard `SQLData` implementation or an Oracle-specific `OracleData` implementation.

- Numeric types

  This includes anything stored in the database as the `NUMBER` SQL type. You use the JPublisher `-numbertypes` option to specify type-mapping for numeric types.

- Large object (LOB) types

  This includes the SQL types, `BLOB` and `CLOB`. You use the JPublisher `-lobtypes` option to specify type-mapping for LOB types.

- Built-in types

  This includes anything stored in the database as a SQL type not covered by the preceding categories. For example, `CHAR`, `VARCHAR2`, `LONG`, and `RAW`. You use the JPublisher `-builtintypes` option to specify type-mapping for built-in types.

Type-Mapping Modes

JPublisher defines the following type-mapping modes, two of which apply to numeric types only:

- JDBC mapping (setting `jdbc`)

  Uses standard default mappings between SQL types and Java native types. For a custom object class, uses a `SQLData` implementation.

- Oracle mapping (setting `oracle`)

  Uses corresponding `oracle.sql` types to map to SQL types. For a custom object, reference, or collection class, uses an `OracleData` implementation.

- Object-JDBC mapping (setting `objectjdbc`)

  Is an extension of the JDBC mapping. Where relevant, object-JDBC mapping uses numeric object types from the standard `java.lang` package, such as `java.lang.Integer`, `Float`, and `Double`, instead of primitive Java types, such as

int, float, and double. The java.lang types are nullable, while the primitive types are not.

- BigDecimal mapping (setting bigdecimal)

  Uses java.math.BigDecimal to map to all numeric attributes. This is appropriate if you are dealing with large numbers but do not want to map to the oracle.sql.NUMBER class.

  Note: Using BigDecimal mapping can significantly degrade performance.

Mapping the Oracle object type to Java

Use the JPublisher -usertypes option to determine how JPublisher will implement the custom Java class that corresponds to an Oracle object type:

- A setting of -usertypes=oracle, which is the default setting, instructs JPublisher to create an OracleData implementation for the custom object class. This will also result in JPublisher producing an OracleData implementation for the corresponding custom reference class.
- A setting of -usertypes=jdbc instructs JPublisher to create a SQLData implementation for the custom object class. No custom reference class can be created. You must use java.sql.Ref, oracle.jdbc.OracleRef, or oracle.sql.REF for the reference type.

  Note: You can also use JPublisher with a -usertypes=oracle setting in creating OracleData implementations to map SQL collection types.

  The -usertypes=jdbc setting is not valid for mapping SQL collection types. The SQLData interface is intended only for mapping Oracle object types.

Mapping Attribute Types to Java

If you do not specify mappings for the attribute types of the Oracle object type, then JPublisher uses the following defaults:

- For numeric attribute types, the default mapping is object-JDBC.
- For LOB attribute types, the default mapping is Oracle.
- For built-in type attribute types, the default mapping is JDBC.

If you want alternate mappings, then use the -numbertypes, -lobtypes, and -builtintypes options, as necessary, depending on the attribute types you have and the mappings you desire.

If an attribute type is itself an Oracle object type, then it will be mapped according to the -usertypes setting.

Important: Be aware that if you specify an SQLData implementation for the custom object class and want the code to be portable, then you must be sure to use portable mappings for the attribute types. The defaults for numeric types and built-in types are portable, but for LOB types you must specify -lobtypes=jdbc.

Summary of SQL Type Categories and Mapping Settings

Table 13–1 summarizes JPublisher categories for SQL types, the mapping settings relevant for each category, and the default settings.

Table 13–1 JPublisher SQL Type Categories, Supported Settings, and Defaults

SQL Type Category	JPublisher Mapping Option	Mapping Settings	Default
UDT types	-usertypes	oracle, jdbc	oracle
numeric types	-numbertypes	oracle, jdbc, objectjdbc, bigdecimal	objectjdbc
LOB types	-lobtypes	oracle, jdbc	oracle
built-in types	-builtintypes	oracle, jdbc	jdbc

Describing an Object Type

Oracle JDBC includes functionality to retrieve information about a structured object type regarding its attribute names and types. This is similar conceptually to retrieving information from a result set about its column names and types, and in fact uses an almost identical method.

This section covers the following topics:

- Functionality for Getting Object Metadata
- Steps for Retrieving Object Metadata

Functionality for Getting Object Metadata

The `oracle.sql.StructDescriptor` class includes functionality to retrieve metadata about a structured object type. The `StructDescriptor` class has a `getMetaData` method with the same functionality as the standard `getMetaData` method available in result set objects. It returns a set of attribute information, such as attribute names and types. Call this method on a `StructDescriptor` object to get metadata about the Oracle object type that the `StructDescriptor` object describes.

The signature of the `StructDescriptor` class `getMetaData` method is the same as the signature specified for `getMetaData` in the standard `ResultSet` interface. The signature is as follows:

```
ResultSetMetaData getMetaData() throws SQLException
```

However, this method actually returns an instance of `oracle.jdbc.StructMetaData`, a class that supports structured object metadata in the same way that the standard `java.sql.ResultSetMetaData` interface specifies support for result set metadata.

The following method is also supported by `StructMetaData`:

```
String getOracleColumnClassName(int column) throws SQLException
```

This method returns the fully qualified name of the `oracle.sql.Datum` subclass whose instances are manufactured if the `OracleResultSet` interface `getOracleObject` method is called to retrieve the value of the specified attribute. For example, `oracle.sql.NUMBER`.

To use the `getOracleColumnClassName` method, you must cast the `ResultSetMetaData` object, which that was returned by the `getMetaData` method, to `StructMetaData`.

> **Note:** In all the preceding method signatures, `column` is something of a misnomer. Where you specify a value of 4 for `column`, you really refer to the fourth attribute of the object.

Steps for Retrieving Object Metadata

Use the following steps to obtain metadata about a structured object type:

1. Create or acquire a `StructDescriptor` instance that describes the relevant structured object type.
2. Call the `getMetaData` method on the `StructDescriptor` instance.
3. Call the metadata getter methods, `getColumnName`, `getColumnType`, and `getColumnTypeName`, as desired.

> **Note:** If one of the structured object attributes is itself a structured object, repeat steps 1 through 3.

Example

The following method shows how to retrieve information about the attributes of a structured object type. This includes the initial step of creating a `StructDescriptor` instance.

```
//
// Print out the ADT's attribute names and types
//
void getAttributeInfo (Connection conn, String type_name) throws SQLException
{
  // get the type descriptor
  StructDescriptor desc = StructDescriptor.createDescriptor (type_name, conn);

  // get type metadata
  ResultSetMetaData md = desc.getMetaData ();

  // get # of attrs of this type
  int numAttrs = desc.length ();

  // temporary buffers
  String attr_name;
  int attr_type;
  String attr_typeName;

  System.out.println ("Attributes of "+type_name+" :");
  for (int i=0; i<numAttrs; i++)
  {
    attr_name = md.getColumnName (i+1);
    attr_type = md.getColumnType (i+1);
    System.out.println (" index"+(i+1)+" name="+attr_name+" type="+attr_type);

    // drill down nested object
    if (attrType == OracleTypes.STRUCT)
    {
      attr_typeName = md.getColumnTypeName (i+1);

      // recursive calls to print out nested object metadata
      getAttributeInfo (conn, attr_typeName);
```

```
        }
    }
}
```

14

Working with LOBs and BFILEs

This chapter describes how to use Java Database Connectivity (JDBC) to access and manipulate large objects (LOB) using either the data interface or the locator interface.

In previous releases, Oracle JDBC drivers required Oracle extensions to standard JDBC types to perform many operations in the Oracle Database. JDBC 3.0 reduced the requirement of using Oracle extensions and JDBC 4.0 nearly eliminated this limitation. Refer to the Javasoft Javadoc for the `java.sql` and `javax.sql` packages, and to the Oracle JDBC Javadoc for details on Oracle extensions.

This chapter contains the following sections:

The LOB Data Types

Prior to Oracle Database 10*g*, the maximum size of a LOB was 2^32 bytes. This restriction has been removed since Oracle Database 10*g*, and the maximum size is limited to the size of available physical storage.

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

The Oracle database supports the following four LOB data types:

- Binary large object (BLOB)

  This data type is used for unstructured binary data.

- Character large object (CLOB)

  This data type is used for character data.

- National character large object (NCLOB)

  This data type is used for national character data.

- BFILE

This data type is used for large binary data objects stored in operating system files, outside of database tablespaces.

BLOBs, CLOBs, and NCLOBs are stored persistently in a database tablespace and all operations performed on these data types are under transaction control.

BFILE is an Oracle proprietary data type that provides read-only access to data located outside the database tablespaces on tertiary storage devices, such as hard disks, network mounted files systems, CD-ROMs, PhotoCDs, and DVDs. BFILE data is not under transaction control and is not stored by database backups.

The PL/SQL language supports the LOB data types and the JDBC interface allows passing IN parameters to PL/SQL procedures or functions, and retrieval of OUT parameters or returns. PL/SQL uses value semantics for all data types including LOBs, but reference semantics only for BFILE.

Oracle SecureFiles

Oracle Database 11*g* Release 1 (11.1) introduced Oracle SecureFiles, a completely new storage for LOBs.

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

Following Features of Oracle SecureFiles are transparently available to JDBC programs through the existing APIs:

- SecureFile compression enables users to compress data to save disk space.
- SecureFile encryption introduces a new encryption facility that allows for random reads and writes of the encrypted data.
- Deduplication enables Oracle database to automatically detect duplicate LOB data and conserve space by storing only one copy of data.
- LOB data path optimization includes logical cache above storage layer and new caching modes.
- High performance space management.

The `setLobOptions` and `getLobOptions` APIs are described in the PL/SQL Packages and Types Reference, and may be accessed from JDBC through callable statements.

Following Oracle SecureFiles features are implemented in the database through updates to the existing APIs:

- isSecureFile Method
- Zero-Copy I/O for Oracle SecureFiles

isSecureFile Method

You can check whether or not your `BLOB` or `CLOB` data uses Oracle SecureFile storage. To achieve this, use the following method from `oracle.jdbc.OracleBlob` or `oracle.jdbc.OracleClob` class:

```
public boolean isSecureFile() throws SQLException
```

If this method returns `true`, then your data uses SecureFile storage.

Zero-Copy I/O for Oracle SecureFiles

With the release of Oracle Database 12*c* Release 1 (12.1) JDBC Drivers, the performance of Oracle SecureFiles operations is greatly improved because Oracle Net Services now uses zero-copy I/O framework for better buffer management.

Oracle Database 11*g* Release 2 introduced a new connection property `oracle.net.useZeroCopyIO`. This property can be used to enable or disable the zero-copy I/O protocol. This connection property is defined as the following constant: `OracleConnection.CONNECTION_PROPERTY_THIN_NET_USE_ZERO_COPY_IO`. If you want to disable the zero-copy I/O framework, then set the value of this connection property to `false`. By default, the value of this connection property is `true`.

Data Interface for LOBs

This section describes the following topics:

- Streamlined Mechanism
- Input
- Output
- CallableSatement and IN OUT Parameter
- Size Limitations

Streamlined Mechanism

The Oracle Database 12*c* Release 1 (12.1) JDBC drivers provide a streamlined mechanism for writing and reading the entire LOB contents. This is referred to as the data interface. The data interface uses standard JDBC methods such as `getString` and `setBytes` to read and write LOB data. It is simpler to code and faster in many cases. Unlike the standard `java.sql.Blob`, `java.sql.Clob` and `java.sql.NClob` interfaces, it does not provide random access capability, that is, it does not use LOB locator and cannot access data beyond 2147483648 elements.

Input

In Oracle Database 12*c* Release 1 (12.1), the `setBytes`, `setBinaryStream`, `setString`, `setCharacterStream`, and `setAsciiStream` methods of `PreparedStatement` are extended to enhance the ability to work with `BLOB`, `CLOB`, and `NCLOB` target columns.

Note: This enhancement does not affect the `BFILE` data because it is read-only.

For the JDBC Oracle Call Interface (OCI) and Thin drivers, there is no limitation on the size of the `byte` array or `String`, and no limitation on the length specified for the stream functions, except the limits imposed by the Java language.

Note: In Java, the array size is limited to positive Java `int` or 2147483648 elements.

For the server-side internal driver, currently there is a limitation of 32767 bytes for operations on SQL statements, such as an `INSERT` statement. This limitation does not

apply for PL/SQL statements. There is a simple workaround for an `INSERT` statement, where it is wrapped in a PL/SQL block in the following way:

```
BEGIN
 INSERT id, c INTO clob_tab VALUES(?,?);
END;
```

You must bear in mind the following automatic switching of the input mode for large data:

- There are three input modes as follows:
  - Direct binding

    This binding is limited in size but most efficient. It places the data for all input columns inline in the block of data sent to the server. All data, including multiple executions of a batch, is sent in a single network operation.
  - Stream binding

    This binding places data at the end. It limits batch size to one and may require multiple round trips to complete.
  - LOB binding

    This binding creates a temporary LOB, copies data to the LOB, and binds the LOB locator. The temporary LOB is automatically freed after execution. The operation to create the temporary LOB and then to writing to the LOB requires multiple round trips. The input of the locators may be batched.
- For SQL statements:
  - The `setBytes` and `setBinaryStream` methods use direct binding for data less than 32767 bytes.
  - The `setBytes` and `setBinaryStream` methods use stream binding for data larger than 32767 bytes.
  - In JDBC 4.0 has introduced new forms of the `setAsciiStream`, `setBinaryStream`, and `setCharacterStream` methods. The form, where the methods take a long argument as length, uses LOB binding for length larger than 2147483648. The form, where the length is not specified, always uses LOB binding.
  - The `setString`, `setCharacterStream`, and `setAsciiStream` methods use direct binding for data smaller than 32767 characters.
  - The `setString`, `setCharacterStream`, and `setAsciiStream` methods use stream binding for data larger than 32766 characters.
  - The new form of `setCharacterStream` method, which takes a `long` argument for length, uses LOB binding for length larger than 2147483647, in JDBC 4.0. The form, where the length is not specified, always uses LOB binding.
- PL/SQL statements
  - The `setBytes` and `setBinary` stream methods use direct binding for data less than 32767 bytes.

Note: If the underlying Database is Oracle Database release 10.x, then this data size limit is 32512 bytes, though you are working with Oracle Database 12*c* Release 1 (12.1) JDBC drivers.

- The `setBytes` and `setBinaryStream` methods use LOB binding for data larger than 32766 bytes.
- The `setString`, `setCharacterStream`, and `setAsciiStream` methods use direct binding for data smaller than 32767 bytes in the database character set.

> **Note:** If the underlying Database is Oracle Database release 10.x, then this data size limit is 32512 bytes, though you are working with Oracle Database 12*c* Release 1 (12.1) JDBC drivers.

- The `setString`, `setCharacterStream`, and `setAsciiStream` methods use LOB binding for data larger than 32766 bytes in the database character set.

The automatic switching of the input mode for large data has impact on certain programs. Previously, you used to get `ORA-17157` errors for attempts to use `setString` method for `String` values larger than 32766 characters. Now, depending on the type of the target parameter, an error may occur while the statement is executed or the operation may succeed.

Another impact is that the automatic switching may result in additional server-side parsing to adapt to the change in the parameter type. This would result in a performance effect, if the data sizes vary above and below the limit for repeated executions of the statement. Switching to the stream modes will effect batching as well.

Forcing conversion to LOB

The `setBytesForBlob` and `setStringForClob` methods, present in the `oracle.jdbc.OraclePreparedStatement` interface, use LOB binding for any data size.

The `SetBigStringTryClob` connection property of Oracle Database 10*g* Release 1 is no longer used or needed.

Output

The `getBytes`, `getBinaryStream`, `getString`, `getCharacterStream`, and `getAsciiStream` methods of `ResultSet` and `CallableStatement` are extended to work with `BLOB`, `CLOB`, and `BFILE` columns or `OUT` parameters. These methods work for any `LOB` of length less than 2147483648.

> **Note:** The `getString` and `getNString` methods cannot be used for retrieving BLOB column values. For more information about `getNString` method, refer to New Methods for National Character Set Type Data in JDK 6 on page 19-4.

The data interface operates by accessing the `LOB` locators within the driver and is transparent to application programming. It works with any supported version of the database, that is, Oracle Database 10.1.x and later. For Oracle Database 11*g* Release 1 or later versions, LOB prefetching may be used to reduce or eliminate any additional database round trips required. For more information, refer to LOB prefetching on page 14-7.

You can read `BFILE` data and read and write `BLOB` or `CLOB` data using the `defineColumnType` method. To read, use `defineColumnType(nn, Types.LONGVARBINARY)` or `defineColumnType(nn,Types.LONGVARCHAR)` method on the column. This produces a direct stream on the data as if it were a `LONG RAW` or `LONG` column. This technique is limited to Oracle Database 10*g* release 1 (10.1) and later.

CallableSatement and IN OUT Parameter

It is a PL/SQL requirement that the Java types used as input and output for an IN OUT parameter must be the same. The automatic switching of types done by the extensions described in this chapter may cause problems with this.

Consider that you have an `IN OUT CLOB` parameter of a stored procedure and you wish to use `setString` method for setting the value for this parameter. For any `IN` and `OUT` parameter, the binds must be of the same type. The automatic switching of the input mode will cause problems unless you are sure of the data sizes. For example, if it is known that neither the input nor output data will ever be larger than 32766 bytes, then you could use `setString` method for the input parameter and register the `OUT` parameter as `Types.VARCHAR` and use `getString` method for the output parameter.

A better solution is to change the stored procedure to have separate `IN` and `OUT` parameters. That is, if you have:

```
CREATE PROCEDURE clob_proc( c IN OUT CLOB );
```

then, change it to:

```
CREATE PROCEDURE clob_proc( c_in IN CLOB, c_out OUT CLOB );
```

Another workaround is to use a container block to make the call. The `clob_proc` procedure can be wrapped with a Java String to use for the `prepareCall` statement, as follows:

```
"DECLARE c_temp; BEGIN c_temp := ?; clob_proc( c_temp); ? := c_temp; END;"
```

In either case, you may use the `setString` method on the first parameter and the `registerOutParameter` method with `Types.CLOB` on the second.

Size Limitations

Be aware of the effect on the performance of the Java memory management system due to creation of very large `byte` array or `String`. Read the information provided by your Java virtual machine (JVM) vendor about the impact of very large data elements on memory management, and consider using the stream interfaces instead.

LOB Locator Interface

Locators are small data structures, which contain information that may be used to access the actual data of the LOB. In a database table, the locator is stored directly in the table, while the data may be in the table or in separate storage. It is common to use separate tablespaces for large LOBs.

In JDBC 4.0, LOBs should be read or written using the interfaces `java.sql.Blob`, `java.sql.Clob`, and `java.sql.NClob`. These provide random access to the data in the LOB.

The Oracle implementation classes `oracle.sql.BLOB`, `oracle.sql.CLOB`, and `oracle.sql.NCLOB` store the locator and access the data with it. The `oracle.sql.BLOB` and `oracle.sql.CLOB` classes implement the `java.sql.Blob` and `java.sql.Clob` interfaces respectively. In `ojdbc6.jar`, `oracle.sql.NCLOB` implements `java.sql.NClob`, but in `ojdbc5.jar`, it implements the `java.sql.Clob` interface.

> **Note:** Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.BLOB` and `oracle.sql.CLOB` classes are deprecated and replaced with the `oracle.jdbc.OracleBlob` and `oracle.jdbc.OracleClob` interfaces. Oracle recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about these interface.

In Oracle Database 12*c* Release 1 (12.1), the Oracle JDBC drivers support the JDBC 4.0 `java.sql.NClob` interface in `ojdbc6.jar` and `ojdbc7.jar`, which are compiled with JDK 6 (must be used with JRE 6) and JDK 7 (must be used with JRE 7) respectively.

In contrast, `oracle.sql.BFILE` is an Oracle extension, without a corresponding `java.sql` interface.

> **See Also:** The JDBC Javadoc for more details.

LOB prefetching

For Oracle Database 12*c* Release 1 (12.1) JDBC drivers, the number of round trips is reduced by prefetching the metadata such as the LOB length and the chunk size as well as the beginning of the LOB data along with the locator during regular fetch operations. If you select LOB columns into a result set, some or all of the data is prefetched to the client, when the locator is fetched. It saves the first roundtrip to retrieve data by deferring all preceding operations until fetching from the locator.

> **Note:** LOB Prefetching is inversely proportional to the size of the LOB data, that is, the benefits of prefetching are more for small LOBs and less for larger LOBs.

The prefetch size is specified in bytes for BLOBs and in characters for CLOBs. It can be specified by setting the connection property `oracle.jdbc.defaultLobPrefetchSize`. The value of this property can be overridden in the following two ways:

- At the statement level: By using the `oracle.jdbc.OracleStatement.setLobPrefetchSize(int)` method
- At the column level: By using the form of `defineColumnType` method that takes length as argument

The default prefetch size is 4000.

> **Note:** Be aware of the possible memory consumption while setting large LOB prefetch sizes in combination with a large row prefetch size and a large number of LOB columns.

> **See Also:** The JDBC Javadoc for more details

New LOB APIs in JDBC 4.0

Oracle Database 11*g* Release 1 introduced the `java.sql.NClob` interface. The Oracle drivers implement the `oracle.sql.NCLOB` and `java.sql.NCLOB` interface in both `ojdbc6.jar` and `ojdbc7.jar`.

The Oracle drivers implement the new factory methods, `createBlob`, `createClob`, and `createNClob` in the `java.sql.Connection` interface to create temporary LOBs.

Starting from JDK 6, the `java.sql.Blob`, `java.sql.Clob`, and `java.sql.NClob` interfaces have a new method `free` to free an LOB and release the associated resources. The Oracle drivers use this method to free an LOB, if it is a temporary LOB.

Working With Temporary LOBs

You can use temporary LOBs to store transient data. The data is stored in temporary table space rather than regular table space. You should free temporary LOBs after you no longer need them. If you do not, then the space the LOB consumes in temporary table space will not be reclaimed.

You can insert temporary LOBs into a table. When you do this, a permanent copy of the LOB is created and stored.

Note: Inserting a temporary LOB may be preferable in some situations. For example, when the LOB data is relatively small and the overhead of copying the data is less than the cost of a database round trip to retrieve the empty locator. Remember that the data is initially stored in the temporary table space on the server and then moved into permanent storage.

You create a temporary LOB with the `static` method `createTemporary`, defined in both the `oracle.sql.BLOB` and `oracle.sql.CLOB` classes. You free a temporary LOB with the `freeTemporary` method.

You can also create a temporary LOB/CLOB or NCLOB by using the connection factory methods available in JDBC 4.0. For more information, refer to "LOB Creation" on page 3-10.

You can test whether a LOB is temporary or not by calling the `isTemporary` method. If the LOB was created by calling the `createTemporary` method, then the `isTemporary` method returns `true`, else it returns `false`.

You can free a temporary LOB by calling the `freeTemporary` method. Free any temporary LOBs before ending the session or call.

Notes:

- If you do not free a temporary LOB, then it will make the storage used by that LOB in the database unavailable. Frequent failure to free temporary LOBs will result in filling up temporary table space with unavailable LOB storage.
- When fetching data from a `ReultSet` with columns that are temporary LOBs, use `getClob` or `getBlob` methods instead of `getString` or `getBytes`.
- The JDBC 4.0 method `free`, present in `java.sql.Blob`, `java.sql.Clob`, and `java.sql.NClob` interfaces, supercedes the `freeTemporary` method.

Opening Persistent LOBs with the Open and Close Methods

This section discusses how to open and close your LOBs. The JDBC implementation of this functionality is provided using the following methods of `oracle.sql.BLOB` and `oracle.sql.CLOB` interfaces:

Note:

- Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.BLOB` and `oracle.sql.CLOB` classes are deprecated and replaced with the `oracle.jdbc.OracleBlob` and `oracle.jdbc.OracleClob` interfaces. Oracle recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about these interface.
- You do not have to necessarily open and close your LOBs. You may choose to open and close them for performance reasons.

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

- `void open (int mode)`
- `void close()`
- `boolean isOpen()`

If you do not wrap LOB operations inside an `Open/Close` call operation, then each modification to the LOB will implicitly open and close the LOB, thereby firing any triggers on a domain index. Note that in this case, any domain indexes on the LOB will become updated as soon as LOB modifications are made. Therefore, domain LOB indexes are always valid and may be used at any time within the same transaction.

If you wrap your LOB operations inside the `Open/Close` call operation, then triggers will not be fired for each LOB modification. Instead, the trigger on domain indexes will be fired at the `Close` call. For example, you might design your application so that domain indexes are not be updated until you call the `close` method. However, this means that any domain indexes on the LOB will not be valid in-between the `Open/Close` calls.

You open a LOB by calling the `open` or `open(int)` method. You may then read and write the LOB without any triggers associated with that LOB firing. When you finish accessing the LOB, close the LOB by calling the `close` method. When you close the LOB, any triggers associated with the LOB will fire.

You can check if a LOB is open or closed by calling the `isOpen` method. If you open the LOB by calling the `open(int)` method, then the value of the argument must be either `MODE_READONLY` or `MODE_READWRITE`, as defined in the `oracle.sql.BLOB` and `oracle.sql.CLOB` classes. If you open the LOB with `MODE_READONLY`, then any attempt to write to the LOB will result in a SQL exception.

Note:

- An error occurs if you commit the transaction before closing all LOBs that were opened by the transaction. The openness of the open LOBs is discarded, but the transaction is successfully committed. Hence, all the changes made to the LOB and non-LOB data in the transaction are committed, but the triggers for domain indexing are not fixed.
- The `open` and `close` methods apply only to persistent LOBs. The `close` method is not similar to the `free` or `freeTemporary` methods used for temporary LOBs. The `free` and `freeTemporary` methods release storage and make a LOB unusable. On the other hand, the `close` method indicates to the database that a modification on a LOB is complete, and triggers should be fired and indexes should be updated. A LOB is still usable after a call to the `close` method.

Working with BFILEs

This section describes how to read data from BFILEs, using file locators. This section covers the following topics:

- Retrieving BFILE Locators
- Writing to BFILES

Retrieving BFILE Locators

The BFILE data type and `oracle.sql.BFILE` classes are Oracle proprietary. So, there is no standard interface for them. You must use Oracle extensions for this type of data.

If you have a standard JDBC result set or callable statement object that includes BFILE locators, then you can access the locators by using the standard result set `getObject` method. This method returns an `oracle.sql.BFILE` object.

You can also access the locators by casting your result set to `OracleResultSet` or your callable statement to `OracleCallableStatement` and using the `getOracleObject` or `getBFILE` method.

Note: If you are using `getObject` or `getOracleObject` methods, then remember to cast the output, as necessary.

Once you have a locator, you can access the BFILE data via the API in `oracle.sql.BFILE`. These APIs are similar to the read methods of the `java.sql.BLOB` interface.

Writing to BFILES

You cannot write data to the contents of the BFILE, but you can use an instance of `oracle.sql.BFILE` as input to a SQL statement or to a PL/SQL procedure. You can achieve this by performing one of the following:

- Use the standard `setObject` method.
- Cast the statement to `OraclePreparedStatement` or `OracleCallableStatement`, and use the `setOracleObject` or `setBFILE` method. These methods take the parameter index and an `oracle.sql.BFILE` object as input.

Note:

- There is no standard `java.sql` interface for BFILEs.
- Use the `getBFILE` methods in the `OracleResultSet` and `OracleCallableStatement` interfaces to retrieve an `oracle.sql.BFILE` object. The `setBFILE` methods in `OraclePreparedStatement` and `OracleCallableStatement` interfaces accept `oracle.sql.BFILE` object as an argument. Use these methods to write to a BFILE.
- Oracle recommends that you use the `getBFILE`, `setBFILE`, and `updateBFILE` methods instead of the `getBfile`, `setBfile`, and `updateBfile` methods. For example, use the `setBFILE` method instead of the `setBfile` method.

BFILEs are read-only. The body of the data resides in the operating system (OS) file system and can be written to using only OS tools and commands. You can create a BFILE for an existing external file by executing the appropriate SQL statement either from JDBC or by using any other way to execute SQL. However, you cannot create an OS file that a BFILE would refer to by SQL or JDBC. Those are created only externally by a process that has access to server file systems.

Note: The code examples present in this chapter, in the earlier versions of this guide, have been removed in favor of references to the sample code available for download on OTN.

15
Using Oracle Object References

This chapter describes the standard Java Database Connectivity (JDBC) that let you access and manipulate object references.

This section discusses the following topics:

Oracle Extensions for Object References

Oracle supports the use of references to database objects. Oracle JDBC provides support for object references as:

- Columns in a `SELECT` clause
- IN or OUT bind variables
- Attributes in an Oracle object
- Elements in a collection type object

In SQL, an object reference (`REF`) is strongly typed. For example, a reference to an `EMPLOYEE` object would be defined as an `EMPLOYEE REF`, not just a `REF`.

When you select an object reference, be aware that you are retrieving only a pointer to an object, not the object itself. You have the choice of materializing the reference as a `java.sql.Ref` instance for portability, or materializing it as an instance of a custom Java class that you have created in advance, which is strongly typed. Custom Java classes used for object references are referred to as **custom reference classes** and must implement the `oracle.jdbc.OracleData` interface.

You can retrieve a `REF` instance through a result set or callable statement object, and pass an updated `REF` instance back to the database through a prepared statement or callable statement object. The `REF` class includes functionality to get and set underlying object attribute values, and get the SQL base type name of the underlying object.

Custom reference classes include this same functionality, as well as having the advantage of being strongly typed. This can help you find coding errors during compilation that might not otherwise be discovered until run time.

Note:

- If you are using the `oracle.jdbc.OracleData` interface for custom object classes, then you will presumably use `OracleData` for corresponding custom reference classes as well. However, if you are using the standard `java.sql.SQLData` interface for custom object classes, then you can only use weak Java types for references. The `SQLData` interface is for mapping SQL object types only.
- You can create and retrieve `REF` objects in your JDBC application only by running SQL statements. There is no JDBC-specific functionality for creating and retrieving `REF` objects.
- You cannot have a reference to an array, even though arrays, like objects, are structured types.

Retrieving and Passing an Object Reference

This section discusses JDBC functionality for retrieving and passing object references. It covers the following topics:

- Retrieving an Object Reference from a Result Set
- Retrieving an Object Reference from a Callable Statement
- Passing an Object Reference to a Prepared Statement

Retrieving an Object Reference from a Result Set

To demonstrate how to retrieve object references, the following example first defines an Oracle object type `ADDRESS`, which is then referenced in the `PEOPLE` table:

```
create type ADDRESS as object
   (street_name     VARCHAR2(30),
    house_no        NUMBER);

create table PEOPLE
    (col1 VARCHAR2(30),
     col2 NUMBER,
     col3 REF ADDRESS);
```

The `ADDRESS` object type has two attributes: a street name and a house number. The `PEOPLE` table has three columns: a column for character data, a column for numeric data, and a column containing a reference to an `ADDRESS` object.

To retrieve an object reference, follow these general steps:

1. Use a standard SQL `SELECT` statement to retrieve the reference from a database table `REF` column.
2. Use `getRef` to get the address reference from the result set into an `OracleRef` instance.
3. Let `Address` be the Java custom class corresponding to the SQL object type `ADDRESS`.
4. Add the correspondence between the Java class `Address` and the SQL type `ADDRESS` to your type map.

5. Use the `getObject` method to retrieve the contents of the `Address` reference. Cast the output to `Address`.

The `PEOPLE` database table is defined earlier in this section. The code for the preceding steps, except the step of adding `Address` to the type map, is as follows:

```
ResultSet rs = stmt.executeQuery("SELECT col3 FROM PEOPLE");
while (rs.next())
{
   OracleRef ref = rs.getRef(1);
   Address a = (Address)
ref.getObject();
}
```

> **Note:** In the preceding code, `stmt` is a previously defined statement object.

Retrieving an Object Reference from a Callable Statement

To retrieve an object reference as an `OUT` parameter in PL/SQL blocks, you must register the bind type for your `OUT` parameter.

1. Cast your callable statement to `OracleCallableStatement`, as follows:

   ```
   OracleCallableStatement ocs =
      (OracleCallableStatement)conn.prepareCall("{? = call func()}");
   ```

2. Register the `OUT` parameter with the following form of the `registerOutParameter` method:

   ```
   ocs.registerOutParameter (int param_index, int sql_type, String sql_type_name);
   ```

   param_index is the parameter index and *sql_type* is the SQL type code. The *sql_type_name* is the name of the structured object type that this reference is used for. For example, if the `OUT` parameter is a reference to an `ADDRESS` object, then `ADDRESS` is the *sql_type_name* that should be passed in.

3. Run the call, as follows:

   ```
   ocs.execute();
   ```

Passing an Object Reference to a Prepared Statement

Pass an object reference to a prepared statement in the same way as you would pass any other SQL type. Use either the `setObject` method or the `setREF` method of a prepared statement object.

Use a prepared statement to update an address reference based on `ROWID`, as follows:

```
PreparedStatement pstmt =
   conn.prepareStatement ("update PEOPLE set ADDR_REF = ? where ROWID = ?");
pstmt.setRef (1, addr_ref);
    pstmt.setRowId (2, rowid);
```

Accessing and Updating Object Values Through an Object Reference

You can use the `Ref` object `setObject` method to update the value of an object in the database through an object reference. To do this, you must first retrieve the reference to the database object and create a Java object that corresponds to the database object.

For example, you can use the code in "Retrieving and Passing an Object Reference" on page 15-2, to retrieve the reference to a database ADDRESS object, as follows:

```
ResultSet rs = stmt.executeQuery("SELECT col3 FROM PEOPLE");
if (rs.next())
{
   Ref ref = rs.getRef(1);
   Address a = (Address)ref.getObject();
}
```

Then, you can create a Java Address object that corresponds to the database ADDRESS object. Use the setObject method of the Ref interface to set the value of the database object, as follows:

```
Address addr = new Address(...);
ref.setObject(addr);
```

Here, the setValue method updates the database ADDRESS object immediately.

Custom Reference Classes with JPublisher

This chapter primarily describes the functionality of the java.sql.Ref class, but it is also possible to access Oracle object references through custom Java classes or, more specifically, custom reference classes.

Custom reference classes offer all the functionality described earlier in this chapter, as well as the advantage of being strongly typed. A custom reference class must satisfy three requirements:

- It must implement the oracle.jdbc.OracleData interface. Note that the standard JDBC SQLData interface, which is an alternative for custom object classes, is not intended for custom reference classes.
- It, or a companion class, must implement the oracle.jdbc.OracleDataFactory interface, for creating instances of the custom reference class.
- It must provide a way to refer to the object data. JPublisher accomplishes this by using an oracle.sql.REF attribute.

> **Note:** Starting from Oracle Database 12*c* Release 1 (12.1), the oracle.sql.REF class is deprecated and replaced with the oracle.jdbc.OracleRef interface, which is a part of the oracle.jdbc package. Oracle strongly recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the oracle.jdbc.OracleRef interface.

You can create custom reference classes yourself, but the most convenient way to produce them is through the Oracle JPublisher utility. If you use JPublisher to generate a custom object class to map to an Oracle object and you specify that JPublisher use an OracleData implementation, then JPublisher will also generate a custom reference class that implements OracleData and OracleDataFactory and includes an oracle.sql.REF attribute. The OracleData implementation will be used if the JPublisher -usertypes mapping option is set to oracle, which is the default.

Custom reference classes are strongly typed. For example, if you define an Oracle object EMPLOYEE, then JPublisher can generate an Employee custom object class and an

`EmployeeRef` custom reference class. Using `EmployeeRef` instances instead of generic `oracle.sql.REF` instances makes it easier to catch errors during compilation instead of at run time. For example, if you accidentally assign some other kind of object reference into an `EmployeeRef` variable.

Be aware that the standard `SQLData` interface supports only SQL object mappings. For this reason, if you instruct JPublisher to implement the standard `SQLData` interface in creating a custom object class, then JPublisher will *not* generate a custom reference class. In this case, your only option is to use standard `java.sql.Ref` instances or `oracle.sql.REF` instances to map to your object references.

See Also:

- "Using JPublisher to Create Custom Object Classes" on page 13-26
- *Oracle Database JPublisher User's Guide*

16
Working with Oracle Collections

This chapter describes Oracle extensions to standard Java Database Connectivity (JDBC) that let you access and manipulate Oracle collections, which map to Java arrays, and their data. The following topics are discussed:

> **Note:** Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.ARRAY` class is deprecated and replaced with the `oracle.jdbc.OracleArray` interface, which is a part of the `oracle.jdbc` package. Oracle recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleArray` interface.

Oracle Extensions for Collections

An Oracle collection, either a variable array (VARRAY) or a nested table in the database, maps to an array in Java. JDBC 2.0 arrays are used to materialize Oracle collections in Java. The terms collection and array are sometimes used interchangeably. However, collection is more appropriate on the database side and array is more appropriate on the JDBC application side.

Oracle supports only named collections, where you specify a SQL type name to describe a type of collection. JDBC enables you to use arrays as any of the following:

- Columns in a `SELECT` clause
- `IN` or `OUT` bind variables
- Attributes in an Oracle object
- Elements of other arrays

This section covers the following topics:

Choices in Materializing Collections

In your application, you have the choice of materializing a collection as an instance of the `oracle.sql.ARRAY` class, which is weakly typed, or materializing it as an instance of a custom Java class that you have created in advance, which is strongly typed. Custom Java classes used for collections are referred to as custom collection classes. A custom collection class must implement the Oracle `oracle.jdbc.OracleData` interface. In addition, the custom class or a companion class must implement `oracle.jdbc.OracleDataFactory`. The standard `java.sql.SQLData` interface is for mapping SQL object types only.

The `oracle.sql.ARRAY` class implements the standard `java.sql.Array` interface.

The `ARRAY` class includes functionality to retrieve the array as a whole, retrieve a subset of the array elements, and retrieve the SQL base type name of the array elements. However, you cannot write to the array, because there are no setter methods.

Custom collection classes, as with the `ARRAY` class, enable you to retrieve all or part of the array and get the SQL base type name. They also have the advantage of being strongly typed, which can help you find coding errors during compilation that might not otherwise be discovered until run time.

Furthermore, custom collection classes produced by JPublisher offer the feature of being writable, with individually accessible elements.

Note: There is no difference in the code between accessing VARRAYs and accessing nested tables. `ARRAY` class methods can determine if they are being applied to a VARRAY or nested table, and respond by taking the appropriate actions.

See Also: For more information about custom collection classes, see "Custom Collection Classes with JPublisher" on page 16-14.

Creating Collections

Because Oracle supports only named collections, you must declare a particular `VARRAY` type name or nested table type name. VARRAY and nested table are not types themselves, but categories of types.

A SQL type name is assigned to a collection when you create it using the SQL `CREATE TYPE` statement:

```
CREATE TYPE <sql_type_name> AS <datatype>;
```

A VARRAY is an array of varying size. It has an ordered set of data elements, and all the elements are of the same data type. Each element has an index, which is a number corresponding to the position of the element in the VARRAY. The number of elements in a VARRAY is the size of the VARRAY. You must specify a maximum size when you declare the `VARRAY` type. For example:

```
CREATE TYPE myNumType AS VARRAY(10) OF NUMBER;
```

This statement defines myNumType as a SQL type name that describes a VARRAY of NUMBER values that can contain no more than 10 elements.

A nested table is an unordered set of data elements, all of the same data type. The database stores a nested table in a separate table which has a single column, and the type of that column is a built-in type or an object type. If the table is an object type, then it can also be viewed as a multi-column table, with a column for each attribute of the object type. You can create a nested table as follows:

```
CREATE TYPE myNumList AS TABLE OF integer;
```

This statement identifies myNumList as a SQL type name that defines the table type used for the nested tables of the type INTEGER.

Creating Multilevel Collection Types

The most common way to create a new multilevel collection type in JDBC is to pass the SQL CREATE TYPE statement to the execute method of the java.sql.Statement class. The following code creates a one-level nested table, first_level, and a two-levels nested table, second_level:

```
Connection conn = ....                              // make a database
                                                    // connection
Statement stmt = conn.createStatement();            // open a database
                                                    // cursor
stmt.execute("CREATE TYPE first_level AS TABLE OF NUMBER");  // create a nested
                                                    // table of number
stmt.execute("CREATE TYPE second_level AS TABLE OF first_level"); // create a
           // two-levels nested table
...        // other operations here
stmt.close();                                       // release the
                                                    // resource
conn.close();                                       // close the
                                                    // database connection
```

Once the multilevel collection types have been created, they can be used as both columns of a base table as well as attributes of a object type.

Overview of Collection Functionality

You can obtain collection data in an array instance through a result set or callable statement and pass it back as a bind variable in a prepared statement or callable statement.

The oracle.sql.ARRAY class, which implements the standard java.sql.Array interface, provides the necessary functionality to access and update the data of an Oracle collection.

This section covers Array Getter and Setter Methods. Use the following result set, callable statement, and prepared statement methods to retrieve and pass collections as Java arrays.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.ARRAY` class is deprecated and replaced with the `oracle.jdbc.OracleArray` interface, which is a part of the `oracle.jdbc` package. Oracle recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleArray` interface.

Result Set and Callable Statement Getter Methods

The `OracleResultSet` and `OracleCallableStatement` interfaces support `getARRAY` and `getArray` methods to retrieve `ARRAY` objects as output parameters, either as `oracle.sql.ARRAY` instances or `java.sql.Array` instances. You can also use the `getObject` method. These methods take as input a `String` column name or `int` column index.

Prepared and Callable Statement Setter Methods

The `OraclePreparedStatement` and `OracleCallableStatement` classes support `setARRAY` and `setArray` methods to take updated `ARRAY` objects as bind variables and pass them to the database. You can also use the `setObject` method. These methods take as input a `String` parameter name or `int` parameter index as well as an `oracle.sql.ARRAY` instance or a `java.sql.Array` instance.

ARRAY Performance Extension Methods

This section discusses the following topics:

- Accessing oracle.sql.ARRAY Elements as Arrays of Java Primitive Types
- ARRAY Automatic Element Buffering
- ARRAY Automatic Indexing

Accessing oracle.sql.ARRAY Elements as Arrays of Java Primitive Types

The `oracle.sql.ARRAY` class contains methods that return array elements as Java primitive types. These methods enable you to access collection elements more efficiently than accessing them as `Datum` instances and then converting each `Datum` instance to its Java primitive value.

Note: These specialized methods of the `oracle.sql.ARRAY` class are restricted to numeric collections.

Each method using the first signature returns collection elements as an *`XXX`*`[]`, where *`XXX`* is a Java primitive type. Each method using the second signature returns a slice of the collection containing the number of elements specified by `count`, starting at the `index` location.

ARRAY Automatic Element Buffering

Oracle JDBC driver provides public methods to enable and disable buffering of `ARRAY` contents.

The following methods are included with the `oracle.sql.ARRAY` class:

- `setAutoBuffering`
- `getAutoBuffering`

It is advisable to enable auto-buffering in a JDBC application when the `ARRAY` elements will be accessed more than once by the `getAttributes` and `getArray` methods, presuming the `ARRAY` data is able to fit into the Java Virtual Machine (JVM) memory without overflow.

Important: Buffering the converted elements may cause the JDBC application to consume a significant amount of memory.

When you enable auto-buffering, the `oracle.sql.ARRAY` object keeps a local copy of all the converted elements. This data is retained so that a second access of this information does not require going through the data format conversion process.

ARRAY Automatic Indexing

If an array is in auto-indexing mode, then the array object maintains an index table to hasten array element access.

The `oracle.sql.ARRAY` class contains the following methods to support automatic array-indexing:

- `setAutoIndexing`
- `setAutoIndexing`

By default, auto-indexing is not enabled. For a JDBC application, enable auto-indexing for `ARRAY` objects if random access of array elements may occur through the `getArray` and `getResultSet` methods.

Creating and Using Arrays

This section discusses how to create array objects and how to retrieve and pass collections as array objects, including the following topics.

- Creating ARRAY Objects
- Retrieving an Array and Its Elements
- Passing Arrays to Statement Objects

Creating ARRAY Objects

Note: Oracle JDBC does not support the JDBC 4.0 method `createArrayOf` method of `java.sql.Connection` interface. This method only allows anonymous array types, while all Oracle array types are named. Use the Oracle specific method `oracle.jdbc.OracleConnection.createARRAY` instead.

This section describes how to create `ARRAY` objects. This section covers the following topics:

- Steps in Creating ARRAY Objects
- Creating Multilevel Collections

Steps in Creating ARRAY Objects

Starting from Oracle Database 11*g* Release 1, you can use the `createARRAY` factory method of `oracle.jdbc.OracleConnection` interface to create an array object. The factory method for creating arrays has been defined as follows:

```
public ARRAY createARRAY(java.lang.String typeName,java.lang.Object
elements)throws SQLException
```

where, `typeName` is the name of the SQL type of the created object and `elements` is the elements of the created object.

Perform the following to create an array:

1. Create a collection with the `CREATE TYPE` statement as follows:

   ```
   CREATE TYPE elements AS varray(22) OF NUMBER(5,2);
   ```

   The two possibilities for the contents of `elements` are:

   - An array of Java primitives. For example, `int[]`.
   - An array of Java objects, such as *xxx*`[]`, where *xxx* is the name of a Java class. For example, `Integer[]`.

2. Construct the `ARRAY` object by passing the Java string specifying the user-defined SQL type name of the array and a Java object containing the individual elements you want the array to contain.

   ```
   ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql_type_name,
   elements);
   ```

Creating Multilevel Collections

As with single-level collections, the JDBC application can create an `oracle.sql.ARRAY` instance to represent a multilevel collection, and then send the instance to the database. The same `createARRAY` factory method you use to create single-level collections, can be used to create multilevel collections as well. To create a single-level collection, the elements are a one dimensional Java array, while to create a multilevel collection, the elements can be either an array of `oracle.sql.ARRAY[]` elements or a nested Java array or the combinations.

The following code shows how to create collection types with a nested Java array:

```
// prepare the multilevel collection elements as a nested Java array
int[][][] elements = { {{1}, {1, 2}}, {{2}, {2, 3}}, {{3}, {3, 4}} };

// create the ARRAY using the factory method
ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql_type_name, elements);
```

Retrieving an Array and Its Elements

This section first discusses how to retrieve an `ARRAY` instance as a whole from a result set, and then how to retrieve the elements from the `ARRAY` instance. This section covers the following topics:

- Retrieving the Array
- Data Retrieval Methods
- Comparing the Data Retrieval Methods

- Retrieving Elements of a Structured Object Array According to a Type Map
- Retrieving a Subset of Array Elements
- Retrieving Array Elements into an oracle.sql.Datum Array
- Accessing Multilevel Collection Elements

Retrieving the Array

You can retrieve a SQL array from a result set by casting the result set to `OracleResultSet` and using the `getARRAY` method, which returns an `oracle.sql.ARRAY` object. If you want to avoid casting the result set, then you can get the data with the standard `getObject` method specified by the `java.sql.ResultSet` interface and cast the output to `oracle.sql.ARRAY`.

Data Retrieval Methods

Once you have an `ARRAY` object, you can retrieve the data using one of these three overloaded methods of the `oracle.sql.ARRAY` class:

- `getArray`
- `getOracleArray`
- `getResultSet`

Oracle also provides methods that enable you to retrieve all the elements of an array, or a subset.

> **Note:** In case you are working with an array of structured objects, Oracle provides versions of these three methods that enable you to specify a type map so that you can choose how to map the objects to Java.

getOracleArray

The `getOracleArray` method is an Oracle-specific extension that is not specified in the standard `Array` interface. The `getOracleArray` method retrieves the element values of the array into a `Datum[]` array. The elements are of the `oracle.sql.*` data type corresponding to the SQL type of the data in the original array.

For an array of structured objects, this method will use `oracle.jdbc.OracleStruct` instances for the elements.

Oracle also provides a `getOracleArray(`*`index`*`,`*`count`*`) method` to get a subset of the array elements.

getResultSet

The `getResultSet` method returns a result set that contains elements of the array designated by the `ARRAY` object. The result set contains one row for each array element, with two columns in each row. The first column stores the index into the array for that element, and the second column stores the element value. In the case of VARRAYs, the index represents the position of the element in the array. In the case of nested tables, which are by definition unordered, the index reflects only the return order of the elements in the particular query.

Oracle recommends using `getResultSet` when getting data from nested tables. Nested tables can have an unlimited number of elements. The `ResultSet` object returned by the method initially points at the first row of data. You get the contents of the nested

table by using the `next` method and the appropriate `get`*`XXX`* method. In contrast, `getArray` returns the entire contents of the nested table at one time.

The `getResultSet` method uses the default type map of the connection to determine the mapping between the SQL type of the Oracle object and its corresponding Java data type. If you do not want to use the default type map of the connection, another version of the method, `getResultSet(`*`map`*`)`, enables you to specify an alternate type map.

Oracle also provides the `getResultSet(`*`index`*`,`*`count`*`)` and `getResultSet(`*`index`*`,`*`count`*`,`*`map`*`)` methods to retrieve a subset of the array elements.

getArray

The `getArray` method is a standard JDBC method that returns the array elements as a `java.lang.Object`, which you can cast as appropriate. The elements are converted to the Java types corresponding to the SQL type of the data in the original array.

Oracle also provides a `getArray(`*`index`*`,`*`count`*`)` method to retrieve a subset of the array elements.

Comparing the Data Retrieval Methods

If you use `getOracleArray` to return the array elements, then the use by that method of `oracle.sql.Datum` instances avoids the expense of data conversion from SQL to Java. The non-character data inside the instance of a `Datum` class or any of its subclass remains in raw SQL format.

If you use `getResultSet` to return an array of primitive data types, then the JDBC driver returns a `ResultSet` object that contains, for each element, the index into the array for the element and the element value. For example:

```
ResultSet rset = intArray.getResultSet();
```

In this case, the result set contains one row for each array element, with two columns in each row. The first column stores the index into the array and the second column stores the element value.

If the elements of an array are of a SQL type that maps to a Java type, then `getArray` returns an array of elements of this Java type. The return type of the `getArray` method is `java.lang.Object`. Therefore, the result must be cast before it can be used.

```
BigDecimal[] values = (BigDecimal[]) intArray.getArray();
```

Here `intArray` is an `oracle.sql.ARRAY`, corresponding to a VARRAY of type `NUMBER`. The `values` array contains an array of elements of type `java.math.BigDecimal`, because the SQL `NUMBER` data type maps to Java `BigDecimal`, by default, according to Oracle JDBC drivers.

> **Note:** Using `BigDecimal` is a resource-intensive operation in Java. Because Oracle JDBC maps numeric SQL data to `BigDecimal` by default, using `getArray` may impact performance, and is not recommended for numeric collections.

Retrieving Elements of a Structured Object Array According to a Type Map

By default, if you are working with an array whose elements are structured objects, and you use `getArray` or `getResultSet`, then the Oracle objects in the array will be mapped to their corresponding Java data types according to the default mapping. This

is because these methods use the default type map of the connection to determine the mapping.

However, if you do not want default behavior, then you can use the `getArray(map)` or `getResultSet(map)` method to specify a type map that contains alternate mappings. If there are entries in the type map corresponding to the Oracle objects in the array, then each object in the array is mapped to the corresponding Java type specified in the type map. For example:

```
Object[] object = (Object[])objArray.getArray(map);
```

Where `objArray` is an `oracle.sql.ARRAY` object and `map` is a `java.util.Map` object.

If the type map does not contain an entry for a particular Oracle object, then the element is returned as an `oracle.jdbc.OracleStruct` object.

The `getResultSet(map)` method behaves similarly to the `getArray(map)` method.

See Also: "Using a Type Map to Map Array Elements" on page 16-12

Retrieving a Subset of Array Elements

If you do not want to retrieve the entire contents of an array, then you can use signatures of `getArray`, `getResultSet`, and `getOracleArray` that let you retrieve a subset. To retrieve a subset of the array, pass in an index and a count to indicate where in the array you want to start and how many elements you want to retrieve. As previously described, you can specify a type map or use the default type map for your connection to convert to Java types. For example:

```
Object object = arr.getArray(index, count, map);
Object object = arr.getArray(index, count);
```

Similar examples using `getResultSet` are:

```
ResultSet rset = arr.getResultSet(index, count, map);
ResultSet rset = arr.getResultSet(index, count);
```

A similar example using `getOracleArray` is:

```
Datum[] arr = arr.getOracleArray(index, count);
```

Where *arr* is an `oracle.sql.ARRAY` object, *index* is type `long`, *count* is type `int`, and *map* is a `java.util.Map` object.

Note: There is no performance advantage in retrieving a subset of an array, as opposed to the entire array.

Retrieving Array Elements into an oracle.sql.Datum Array

Use `getOracleArray` to return an `oracle.sql.Datum[]` array. The elements of the returned array is of `oracle.sql.*` type that correspond to the SQL data type of the elements of the original array. For example:

```
Datum arraydata[] = arr.getOracleArray();
```

`arr` is an `oracle.sql.ARRAY` object.

The following example assumes that a connection object `conn` and a statement object `stmt` have already been created. In the example, an array with the SQL type name `NUM_ARRAY` is created to store a VARRAY of `NUMBER` data. The `NUM_ARRAY` is in turn stored in a table `VARRAY_TABLE`.

A query selects the contents of the VARRAY_TABLE. The result set is cast to OracleResultSet. The getARRAY method is applied to it to retrieve the array data into my_array, which is an oracle.sql.ARRAY object.

Because my_array is of type oracle.sql.ARRAY, you can apply the methods getSQLTypeName and getBaseType to it to return the name of the SQL type of each element in the array and its integer code.

The program then prints the contents of the array. Because the contents of NUM_ARRAY are of the SQL data type NUMBER, the elements of my_array are of the type, BigDecimal. Before you can use the elements, they must first be cast to BigDecimal. In the for loop, the individual values of the array are cast to BigDecimal and printed to standard output.

```
stmt.execute ("CREATE TYPE num_varray AS VARRAY(10) OF NUMBER(12, 2)");
stmt.execute ("CREATE TABLE varray_table (col1 num_varray)");
stmt.execute ("INSERT INTO varray_table VALUES (num_varray(100, 200))");

ResultSet rs = stmt.executeQuery("SELECT * FROM varray_table");
ARRAY my_array = ((OracleResultSet)rs).getARRAY(1);

// return the SQL type names, integer codes,
// and lengths of the columns
System.out.println ("Array is of type " + array.getSQLTypeName());
System.out.println ("Array element is of type code " + array.getBaseType());
System.out.println ("Array is of length " + array.length());

// get Array elements
BigDecimal[] values = (BigDecimal[]) my_array.getArray();

for (int i=0; i<values.length; i++)
{
   BigDecimal out_value = (BigDecimal) values[i];
   System.out.println(">> index " + i + " = " + out_value.intValue());
}
```

Note that if you use getResultSet to obtain the array, then you must would first get the result set object, and then use the next method to iterate through it. Notice the use of the parameter indexes in the getInt method to retrieve the element index and the element value.

```
ResultSet rset = my_array.getResultSet();
while (rset.next())
{
    // The first column contains the element index and the
    // second column contains the element value
    System.out.println(">> index " + rset.getInt(1)+" = " + rset.getInt(2));
}
```

Accessing Multilevel Collection Elements

The oracle.sql.ARRAY class provides three methods, which are overloaded, to access collection elements. The JDBC drivers extend these methods to support multilevel collections. These methods are:

- getArray method
- getOracleArray method
- getResultSet method

The `getArray` method returns a Java array that holds the collection elements. The array element type is determined by the collection element type and the JDBC default conversion matrix.

For example, the `getArray` method returns a `java.math.BigDecimal` array for collection of SQL `NUMBER`. The `getOracleArray` method returns a `Datum` array that holds the collection elements in `Datum` format. For multilevel collections, the `getArray` and `getOracleArray` methods both return a Java array of `oracle.sql.ARRAY` elements.

The `getResultSet` method returns a `ResultSet` object that wraps the multilevel collection elements. For multilevel collections, the JDBC applications use the `getObject`, `getARRAY`, or `getArray` method of the `ResultSet` class to access the collection elements as instances of `oracle.sql.ARRAY`.

The following code shows how to use the `getOracleArray`, `getArray`, and `getResultSet` methods:

```
Connection conn = ...;          // make a JDBC connection
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery ("select col2 from tab2 where idx=1");

while (rset.next())
{
  ARRAY varray3 = (ARRAY) rset.getObject (1);
  Object varrayElems = varray3.getArray (1);
// access array elements of "varray3"
  Datum[] varray3Elems = (Datum[]) varrayElems;

  for (int i=0; i<varray3Elems.length; i++)
  {
    ARRAY varray2 = (ARRAY) varray3Elems[i];
    Datum[] varray2Elems = varray2.getOracleArray();
    // access array elements of  "varray2"

    for (int j=0; j<varray2Elems.length; j++)
    {
      ARRAY varray1 = (ARRAY) varray2Elems[j];
      ResultSet varray1Elems = varray1.getResultSet();
      // access array elements   of "varray1"

      while (varray1Elems.next())
        System.out.println ("idx="+varray1Elems.getInt(1)+"
          value="+varray1Elems.getInt(2));
    }
  }
}
rset.close ();
stmt.close ();
conn.close ();
```

Passing Arrays to Statement Objects

This section discusses how to pass arrays to prepared statement objects or callable statement objects.

Passing an Array to a Prepared Statement

Pass an array to a prepared statement as follows.

> **Note:** you can use arrays as either `IN` or `OUT` bind variables.

1. Define the array that you want to pass to the prepared statement as an `oracle.sql.ARRAY` object.

   ```
   ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql_type_name,
   elements);
   ```

   `sql_type_name` is a Java string specifying the user-defined SQL type name of the array and *`elements`* is a `java.lang.Object` containing a Java array of the elements.

2. Create a `java.sql.PreparedStatement` object containing the SQL statement to be run.

3. Cast your prepared statement to `OraclePreparedStatement`, and use `setARRAY` to pass the array to the prepared statement.

   ```
   (OraclePreparedStatement)stmt.setARRAY(parameterIndex, array);
   ```

   `parameterIndex` is the parameter index and *`array`* is the `oracle.sql.ARRAY` object you constructed previously.

4. Run the prepared statement.

Passing an Array to a Callable Statement

To retrieve a collection as an `OUT` parameter in PL/SQL blocks, perform the following to register the bind type for your `OUT` parameter.

1. Cast your callable statement to `OracleCallableStatement`, as follows:

   ```
   OracleCallableStatement ocs = (OracleCallableStatement)conn.prepareCall("{? =
   call func()}");
   ```

2. Register the `OUT` parameter with the following form of the `registerOutParameter` method:

   ```
   ocs.registerOutParameter
           (int param_index, int sql_type, string sql_type_name);
   ```

   `param_index` is the parameter index, *`sql_type`* is the SQL type code, and *`sql_type_name`* is the name of the array type. In this case, the *`sql_type`* is `OracleTypes.ARRAY`.

3. Run the call, as follows:

   ```
   ocs.execute();
   ```

4. Get the value, as follows:

   ```
   oracle.sql.ARRAY array = ocs.getARRAY(1);
   ```

Using a Type Map to Map Array Elements

If your array contains Oracle objects, then you can use a type map to associate the objects in the array with the corresponding Java class. If you do not specify a type map, or if the type map does not contain an entry for a particular Oracle object, then each element is returned as an `oracle.jdbc.OracleStruct` object.

If you want the type map to determine the mapping between the Oracle objects in the array and their associated Java classes, then you must add an appropriate entry to the map.

The following example illustrates how you can use a type map to map the elements of an array to a custom Java object class. In this case, the array is a nested table. The example begins by defining an EMPLOYEE object that has a name attribute and employee number attribute. EMPLOYEE_LIST is a nested table type of EMPLOYEE objects. Then an EMPLOYEE_TABLE is created to store the names of departments within a corporation and the employees associated with each department. In the EMPLOYEE_TABLE, the employees are stored in the form of EMPLOYEE_LIST tables.

```
stmt.execute("CREATE TYPE EMPLOYEE AS OBJECT
            (EmpName VARCHAR2(50),EmpNo INTEGER))");

stmt.execute("CREATE TYPE EMPLOYEE_LIST AS TABLE OF EMPLOYEE");

stmt.execute("CREATE TABLE EMPLOYEE_TABLE (DeptName VARCHAR2(20),
     Employees EMPLOYEE_LIST) NESTED TABLE Employees STORE AS ntable1");

stmt.execute("INSERT INTO EMPLOYEE_TABLE VALUES ("SALES", EMPLOYEE_LIST
            (EMPLOYEE('Susan Smith', 123), EMPLOYEE('Lee Brown', 124)))");
```

If you want to retrieve all the employees belonging to the SALES department into an array of instances of the custom object class EmployeeObj, then you must add an entry to the type map to specify mapping between the EMPLOYEE SQL type and the EmployeeObj custom object class.

To do this, first create your statement and result set objects, then select the EMPLOYEE_LIST associated with the SALES department into the result set. Cast the result set to OracleResultSet so you can use the getARRAY method to retrieve the EMPLOYEE_LIST into an ARRAY object (employeeArray in the following example).

The EmployeeObj custom object class in this example implements the SQLData interface.

```
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)s.executeQuery
       ("SELECT Employees FROM employee_table WHERE DeptName = 'SALES'");

// get the array object
ARRAY employeeArray = ((OracleResultSet)rs).getARRAY(1);
```

Now that you have the EMPLOYEE_LIST object, get the existing type map and add an entry that maps the EMPLOYEE SQL type to the EmployeeObj Java type.

```
// add type map entry to map SQL type
// "EMPLOYEE" to Java type "EmployeeObj"
Map map = conn.getTypeMap();
map.put("EMPLOYEE", Class.forName("EmployeeObj"));
```

Next, retrieve the SQL EMPLOYEE objects from the EMPLOYEE_LIST. To do this, call the getArray method of the employeeArray array object. This method returns an array of objects. The getArray method returns the EMPLOYEE objects into the employees object array.

```
// Retrieve array elements
Object[] employees = (Object[]) employeeArray.getArray();
```

Finally, create a loop to assign each of the EMPLOYEE SQL objects to the EmployeeObj Java object emp.

```
// Each array element is mapped to EmployeeObj object.
for (int i=0; i<employees.length; i++)
{
   EmployeeObj emp = (EmployeeObj) employees[i];
   ...
}
```

Custom Collection Classes with JPublisher

This chapter primarily describes the functionality of the `oracle.sql.ARRAY` class, but it is also possible to access Oracle collections through custom Java classes or, more specifically, custom collection classes.

You can create custom collection classes yourself, but the most convenient way is to use the Oracle JPublisher utility. Custom collection classes generated by JPublisher offer all the functionality described earlier in this chapter, as well as the following advantages:

- They are strongly typed. This can help you find coding errors during compilation that might not otherwise be discovered until run time.
- They can be changeable, or mutable. Custom collection classes produced by JPublisher, unlike the `ARRAY` class, allow you to get and set individual elements using the `getElement` and `setElement` methods.

A custom collection class must satisfy three requirements:

- It must implement the `oracle.jdbc.OracleData` interface. Note that the standard JDBC `SQLData` interface, which is an alternative for custom object classes, is not intended for custom collection classes.
- It, or a companion class, must implement the `oracle.jdbc.OracleDataFactory` interface, for creating instances of the custom collection class.
- It must have a means of storing the collection data. Typically it will directly or indirectly include an `oracle.sql.ARRAY` attribute for this purpose.

A JPublisher-generated custom collection class implements `OracleData` and `OracleDataFactory` and indirectly includes an `oracle.sql.ARRAY` attribute. The custom collection class will have an `oracle.jpub.runtime.MutableArray` attribute. The `MutableArray` class has an `oracle.sql.ARRAY` attribute.

Note: When you use JPublisher to create a custom collection class, you must use the `OracleData` implementation. This will be true if the JPublisher `-usertypes` mapping option is set to `oracle`, which is the default.

You cannot use a `SQLData` implementation for a custom collection class. Setting the `-usertypes` mapping option to `jdbc` is invalid.

As an example of custom collection classes being strongly typed, if you define an Oracle collection `MYVARRAY`, then JPublisher can generate a `MyVarray` custom collection class. Using `MyVarray` instances, instead of generic `oracle.sql.ARRAY` instances, makes it easier to catch errors during compilation instead of at run time. For example, if you accidentally assign some other kind of array into a `MyVarray` variable.

If you do not use custom collection classes, then you would use standard `java.sql.Array` instances, or `oracle.sql.ARRAY` instances, to map to your collections.

See Also:

- "Using JPublisher to Create Custom Object Classes" on page 13-26
- *Oracle Database JPublisher User's Guide*

17
Result Set

Standard Java Database Connectivity (JDBC) features in Java Development Kit (JDK) include enhancements to result set functionality, such as processing forward or backward, positioning relatively or absolutely, seeing changes to the database made internally or externally, and updating result set data and then copying the changes to the database.

This chapter discusses the following topics:

Oracle JDBC Implementation Overview for Result Set Support

This section discusses key aspects of the Oracle JDBC implementation of result set support for scrollability, through use of a client-side cache, and for updatability, through use of `ROWID`s.

It is permissible for customers to implement their own client-side caching mechanism, and Oracle provides an interface to use in doing so.

Oracle JDBC Implementation for Result Set Scrollability

Because the underlying server does *not* support scrollable cursors, Oracle JDBC must implement scrollability in a separate layer.

It is important to be aware that this is accomplished by using a client-side memory cache to store rows of a scrollable result set.

Important: Because all rows of any scrollable result set are stored in the client-side cache, a situation, where the result set contains many rows, many columns, or very large columns, might cause the client-side Java Virtual Machine (JVM) to fail. Do not specify scrollability for a large result set.

Oracle JDBC Implementation for Result Set Updatability

To support updatability, Oracle JDBC uses `ROWID` to uniquely identify database rows that appear in a result set. For every query into an updatable result set, Oracle JDBC driver automatically retrieves the `ROWID` along with the columns you select.

Note: Client-side caching is not required by updatability in and of itself. In particular, a forward-only updatable result set will not require a client-side cache.

Resultset Limitations and Downgrade Rules

Some types of result sets are not feasible for certain kinds of queries. If you specify an unfeasible result set type or concurrency type for the query you run, then the JDBC driver follows a set of rules to determine the best feasible types to use instead.

The actual result set type and concurrency type are determined when the statement is run, with the driver issuing a `SQLWarning` on the statement object if the desired result set type or concurrency type is not feasible. The `SQLWarning` object will contain the reason why the requested type was not feasible. Check for warnings to verify whether you received the type of result set that you requested.

Result Set Limitations

The following limitations are placed on queries for enhanced result sets. Failure to follow these guidelines results in the JDBC driver choosing an alternative result set type or concurrency type.

To produce an updatable result set:

- A query can select from only a single table and cannot contain any join operations.

  In addition, for inserts to be feasible, the query must select all non-nullable columns and all columns that do not have a default value.

- A query cannot use `SELECT *`.

  However, there is a workaround for this.

- A query must select table columns only.

  It cannot select derived columns or aggregates, such as the `SUM` or `MAX` of a set of columns.

To produce a scroll-sensitive result set:

- A query cannot use `SELECT *`.

  However, there is a workaround for this.

- A query can select from only a single table.

Scrollable and updatable result sets cannot have any column as `Stream`. When the server has to fetch a `Stream` column, it reduces the fetch size to one and blocks all columns following the `Stream` column until the `Stream` column is read. As a result, columns cannot be fetched in bulk and scrolled through.

Workaround

As a workaround for the `SELECT *` limitation, you can use table aliases, as shown in the following example:

```
SELECT t.* FROM TABLE t ...
```

> **Note:** There is a simple way to determine if your query will probably produce a scroll-sensitive or updatable result set: If you can legally add a ROWID column to the query list, then the query is probably suitable for either a scroll-sensitive or an updatable result set.

Result Set Downgrade Rules

If the specified result set type or concurrency type is not feasible, then Oracle JDBC driver uses the following rules in choosing alternate types:

- If the specified result set type is TYPE_SCROLL_SENSITIVE, but the JDBC driver cannot fulfill that request, then the driver attempts a downgrade to TYPE_SCROLL_INSENSITIVE.
- If the specified or downgraded result set type is TYPE_SCROLL_INSENSITIVE, but the JDBC driver cannot fulfill that request, then the driver attempts a downgrade to TYPE_FORWARD_ONLY.
- If the specified concurrency type is CONCUR_UPDATABLE, but the JDBC driver cannot fulfill that request, then the JDBC driver attempts a downgrade to CONCUR_READ_ONLY.

> **Note:** Any manipulations of the result set type and concurrency type by the JDBC driver are independent of each other.

Verifying Result Set Type and Concurrency Type

After a query has been run, you can verify the result set type and concurrency type that the JDBC driver actually used, by calling methods on the result set object.

- `int getType() throws SQLException`

  This method returns an `int` value for the result set type used for the query. `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_SENSITIVE`, or `ResultSet.TYPE_SCROLL_INSENSITIVE` are the possible values.

- `int getConcurrency() throws SQLException`

  This method returns an `int` value for the concurrency type used for the query. `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE` are the possible values.

Avoiding Update Conflicts

It is important to be aware of the following facts regarding updatable result sets with the JDBC drivers:

- The drivers do not enforce write locks for an updatable result set.
- The drivers do not check for conflicts with a result set DELETE or UPDATE operation.

A conflict will occur if you try to perform a DELETE or UPDATE operation on a row updated by another committed transaction.

Oracle JDBC drivers use the ROWID to uniquely identify a row in a database table. As long as the ROWID is valid when a driver tries to send an UPDATE or DELETE operation to the database, the operation will be run.

The driver will not report any changes made by another committed transaction. Any conflicts are silently ignored and your changes will overwrite the previous changes.

To avoid such conflicts, use the Oracle `FOR UPDATE` feature when running the query that produces the result set. This will avoid conflicts, but will also prevent simultaneous access to the data. Only a single write lock can be held concurrently on a data item.

Fetch Size

By default, when Oracle JDBC runs a query, it retrieves a result set of 10 rows at a time from the database cursor. This is the default Oracle row fetch size value. You can change the number of rows retrieved with each trip to the database cursor by changing the row fetch size value.

Standard JDBC also enables you to specify the number of rows fetched with each database round-trip for a query, and this number is referred to as the fetch size. In Oracle JDBC, the row-prefetch value is used as the default fetch size in a statement object. Setting the fetch size overrides the row-prefetch setting and affects subsequent queries run through that statement object.

Fetch size is also used in a result set. When the statement object run a query, the fetch size of the statement object is passed to the result set object produced by the query. However, you can also set the fetch size in the result set object to override the statement fetch size that was passed to it.

Note: Changes made to the fetch size of a statement object after a result set is produced will have no affect on that result set.

The result set fetch size, either set explicitly, or by default equal to the statement fetch size that was passed to it, determines the number of rows that are retrieved in any subsequent trips to the database for that result set. This includes any trips that are still required to complete the original query, as well as any refetching of data into the result set. Data can be refetched, either explicitly or implicitly, to update a scroll-sensitive or scroll-insensitive/updatable result set.

Setting the Fetch Size

The following methods are available in all `Statement`, `PreparedStatement`, `CallableStatement`, and `ResultSet` objects for setting and getting the fetch size:

- `void setFetchSize(int rows) throws SQLException`
- `int getFetchSize() throws SQLException`

To set the fetch size for a query, call `setFetchSize` on the statement object prior to running the query. If you set the fetch size to N, then N rows are fetched with each trip to the database.

After you have run the query, you can call `setFetchSize` on the result set object to override the statement object fetch size that was passed to it. This will affect any subsequent trips to the database to get more rows for the original query, as well as affecting any later refetching of rows.

Presetting the Fetch Direction

The standard JDBC enables to pre-specify the direction, known as the fetch direction, for use in processing a result set. This allows the JDBC driver to optimize its processing. The following result set methods are specified:

- `void setFetchDirection(int direction) throws SQLException`
- `int getFetchDirection() throws SQLException`

Oracle JDBC drivers support only the forward preset value, which you can specify by entering the `ResultSet.FETCH_FORWARD` static constant value.

The values `ResultSet.FETCH_REVERSE` and `ResultSet.FETCH_UNKNOWN` are not supported. Attempting to specify them causes a SQL warning, and the settings are ignored.

Refetching Rows

The result set `refreshRow` method is supported for some types of result sets for refetching data. This consists of going back to the database to re-obtain the database rows that correspond to *n* rows in the result set, starting with the current row, where *n* is the fetch size. This lets you see the latest updates to the database that were made outside of your result set, subject to the isolation level of the enclosing transaction.

Because refetching re-obtains only rows that correspond to rows already in your result set, it does nothing about rows that have been inserted or deleted in the database since the original query. It ignores rows that have been inserted, and rows will remain in your result set even after the corresponding rows have been deleted from the database. When there is an attempt to refetch a row that has been deleted in the database, the corresponding row in the result set will maintain its original values.

> **Note:** If you declare a `TYPE_SCROLL_SENSITIVE` Result Set based on a query with certain criteria and then externally update the row so that the column values no longer match the query criteria, the driver behaves as if the row has been deleted from the database and the row is not retrieved by the query issued. So, you do not see the updates to the particular row when you call the `refreshRow` method.

Following is the signature of the `refreshRow` method:

```
void refreshRow() throws SQLException
```

You must be at a valid current row when you call this method, not outside the row bounds and not at the insert-row.

The `refreshRow` method is supported for the following result set categories:

- scroll-sensitive/read-only
- scroll-sensitive/updatable
- scroll-insensitive/updatable

> **Note:** Scroll-sensitive result set functionality is implemented through implicit calls to `refreshRow`.

Viewing Database Changes Made Internally and Externally

This section discusses the ability of a result set to view the following:

- Own changes of the result set, referred to as internal changes
- Changes made from elsewhere, either from your own transaction outside the result set, or from other committed transactions, referred to as external changes

> **Note:** External changes are referred to as other's changes in the standard JDBC specification.

This section covers the following topics:

Visibility versus Detection of External Changes

Regarding the changes made to an underlying database by external sources, there are two similar but distinct concepts with respect to visibility of the changes from your local result set:

- Visibility of changes
- Detection of changes

A "visible" change means that when you look at a row in the result set, you can see new data values from changes made by external sources, to the corresponding row in the database.

A "detected" change, however, means that the result set is aware that this is a new value since the result set was first populated.

Even when an Oracle result set sees new data, as with an external `UPDATE` in a scroll-sensitive result set, it has no awareness that this data has changed since the result set was populated. Such changes are not detected.

Summary of Visibility of Internal and External Changes

Table 17–1 summarizes how a result set object in the Oracle JDBC implementation can see changes made internally through the result set itself, and changes made externally to the underlying database from elsewhere in your transaction or from other committed transactions.

Table 17–1 Visibility of Internal and External Changes for Oracle JDBC

Result Set Type	Can See Internal DELETE?	Can See Internal UPDATE?	Can See Internal INSERT?	Can See External DELETE?	Can See External UPDATE?	Can See External INSERT?
forward-only	no	yes	no	no	no	no
scroll-sensitive	yes	yes	no	no	yes	no
scroll-insensitive	yes	yes	no	no	no	no

Note:

- Remember that explicit use of the `refreshRow` method, is distinct from the concept of visibility of external changes.
- Remember that even when external changes are visible, as with `UPDATE` operations underlying a scroll-sensitive result set, they are not detected. The result set `rowDeleted`, `rowUpdated`, and `rowInserted` methods always return `false`.

Oracle Implementation of Scroll-Sensitive Result Sets

The Oracle implementation of scroll-sensitive result sets involves the concept of a window, with a window size that is based on the fetch size. The window size affects how often rows are updated in the result set.

Once you establish a current row by moving to a specified row, the window consists of the *n* rows in the result set starting with that row, where *n* is the fetch size being used by the result set. Note that there is no current row, and therefore no window, when a result set is first created. The default position is before the first row, which is not a valid current row.

As you move from row to row, the window remains unchanged as long as the current row stays within that window. However, once you move to a new current row outside the window, you redefine the window to be the N rows starting with the new current row.

Whenever the window is redefined, the N rows in the database corresponding to the rows in the new window are automatically refetched through an implicit call to the `refreshRow` method, thereby updating the data throughout the new window.

So external updates are not instantaneously visible in a scroll-sensitive result set. They are only visible after the automatic refetches just described.

Note: This kind of refetching is not a highly efficient or optimized methodology and it has significant performance concerns. Consider carefully before using scroll-sensitive result sets as currently implemented. There is also a significant trade-off between sensitivity and performance. The most sensitive result set is one with a fetch size of 1, which would result in the new current row being refetched every time you move between rows. However, this would have a significant impact on the performance of your application.

18
JDBC RowSets

This chapter contains the following sections:

Overview of JDBC RowSets

A RowSet is an object that encapsulates a set of rows from either java Database Connectivity (JDBC) result sets or tabular data sources. RowSets support component-based development models like JavaBeans, with a standard set of properties and an event notification mechanism.

RowSets were introduced in JDBC 2.0 through the optional packages. However, the implementation of RowSets was standardized in the JDBC RowSet Implementations Specification (JSR-114), which is available as non-optional package since Java Platform, Standard Edition (Java SE) 5.0. Java SE 6.0 RowSets contain more APIs supporting features like RowId, National Language Charactersets, and so on. The Java SE Javadocs provide information about the standard interfaces and base classes for JDBC RowSet implementations.

See Also:

- Java SE 6 Javadoc at:
  `http://docs.oracle.com/javase/6/docs/api/`
- Java SE 7 Javadoc at:
  `http://docs.oracle.com/javase/7/docs/api/`

The JSR-114 specification includes implementation details for five types of RowSet:

- `CachedRowSet`
- `JdbcRowSet`
- `WebRowSet`
- `FilteredRowSet`
- `JoinRowSet`

Oracle JDBC supports all five types of RowSets through the interfaces and classes present in the `oracle.jdbc.rowset` package. Since Oracle Database 11*g* Release 1, RowSets support has been added in the server-side drivers. Therefore, starting from Oracle Database 11*g* Release 1, RowSets support is uniform across all Oracle JDBC driver types. The standard Oracle JDBC Java Archive (JAR) files, for example, `ojdbc6.jar` and `ojdbc7.jar` contain the `oracle.jdbc.rowset` package.

Note:

- The other JAR files with different file suffix names, for example, `ojdbc6_g.jar`, `ojdbc6dms.jar`, and so on also contain the `oracle.jdbc.rowset` package.
- In Oracle Database 10*g* release 2, the implementation classes were packaged in the `ojdbc14.jar` file.
- Prior to Oracle Database 10*g* release 2, the implementation classes were packaged in the `ocrs12.jar` file.
- Prior to Oracle Database 11*g* Release 1, RowSets support was not available in the server-side drivers.

To use the Oracle RowSet implementations, you need to import either the entire `oracle.jdbc.rowset` package or specific classes and interfaces from the package for the required RowSet type. For client-side usage, you also need to include the standard Oracle JAR files like `ojdbc6.jar` or `ojdbc7.jar` in the `CLASSPATH` environment variable.

> **See Also:** "Check the Environment Variables" on page 2-3 for information about setting the `CLASSPATH` environment variable.

This section covers the following topics:

- RowSet Properties
- Events and Event Listeners
- Command Parameters and Command Execution
- Traversing RowSets

RowSet Properties

The `javax.sql.RowSet` interface provides a set of JavaBeans properties that can be altered to access the data in the data source through a single interface. Example of properties are connection string, user name, password, type of connection, and the query string.

For a complete list of properties and property descriptions, refer to the Java2 Platform, Standard Edition (J2SE) Javadoc for `javax.sql.RowSet` at
`http://docs.oracle.com/javase/1.5.0/docs/api/javax/sql/RowSet.html`

The interface provides standard accessor methods for setting and retrieving the property values. The following code illustrates setting some of the `RowSet` properties:

```
...
rowset.setUrl("jdbc:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT employee_id, first_name, last_name, salary FROM
```

```
employees");
...
```

In this example, the URL, user name, password, and SQL query are set as the `RowSet` properties to retrieve the employee number, employee name, and salary of all the employees into the `RowSet` object.

Events and Event Listeners

RowSets support JavaBeans events. The following types of events are supported by the `RowSet` interface:

- `cursorMoved`

  This event is generated whenever there is a cursor movement. For example, when the `next` or `previous` method is called.

- `rowChanged`

  This event is generated when a row is inserted, updated, or deleted from the RowSet.

- `rowSetChanged`

  This event is generated when the whole RowSet is created or changed. For example, when the `execute` method is called.

An application component can implement a RowSet listener to listen to these RowSet events and perform desired operations when the event occurs. Application components, which are interested in these events, must implement the standard `javax.sql.RowSetListener` interface and register such listener objects with a `RowSet` object. A listener can be registered using the `RowSet.addRowSetListener` method and unregistered using the `RowSet.removeRowSetListener` method. Multiple listeners can be registered with the same `RowSet` object.

The following code illustrates the registration of a RowSet listener:

```
...
MyRowSetListener rowsetListener = new MyRowSetListener ();
// adding a rowset listener
rowset.addRowSetListener (rowsetListener);
...
```

The following code illustrates a listener implementation:

```
public class MyRowSetListener implements RowSetListener
 {
   public void cursorMoved(RowSetEvent event)
   {
     // action on cursor movement
   }

   public void rowChanged(RowSetEvent event)
   {
     // action on change of row
   }

   public void rowSetChanged(RowSetEvent event)
   {
     // action on changing of rowset
   }
 }// end of class MyRowSetListener
```

Applications that need to handle only selected events can implement only the required event handling methods by using the `oracle.jdbc.rowset.OracleRowSetListenerAdapter` class, which is an abstract class with empty implementation for all the event handling methods. In the following code, only the `rowSetChanged` event is handled, while the remaining events are not handled by the application:

```
...
rowset.addRowSetListener(new oracle.jdbc.rowset.OracleRowSetListenerAdapter ()
   {
     public void rowSetChanged(RowSetEvent event)
     {
       // your action for rowSetChanged
     }
   }
 );
...
```

Command Parameters and Command Execution

The `command` property of a `RowSet` object typically represents a SQL query string, which when processed would populate the `RowSet` object with actual data. Like in regular JDBC processing, this query string can take input or bind parameters. The `javax.sql.RowSet` interface also provides methods for setting input parameters to this SQL query. After the required input parameters are set, the SQL query can be processed to populate the `RowSet` object with data from the underlying data source. The following code illustrates this simple sequence:

```
...
 rowset.setCommand("SELECT first_name, last_name, salary FROM employees WHERE
employee_id = ?");
 // setting the employee number input parameter for employee named "Douglas"
 rowset.setInt(1, 199);
 rowset.execute();
 ...
```

In the preceding example, the employee number 199 is set as the input or bind parameter for the SQL query specified in the `command` property of the `RowSet` object. When the SQL query is processed, the `RowSet` object is filled with the employee name and salary information of the employee whose employee number is 199.

Traversing RowSets

The `javax.sql.RowSet` interface extends the `java.sql.ResultSet` interface. The `RowSet` interface, therefore, provides cursor movement and positioning methods, which are inherited from the `ResultSet` interface, for traversing through data in a `RowSet` object. Some of the inherited methods are `absolute`, `beforeFirst`, `afterLast`, `next`, and `previous`.

The `RowSet` interface can be used just like a `ResultSet` interface for retrieving and updating data. The `RowSet` interface provides an optional way to implement a scrollable and updatable result set. All the fields and methods provided by the `ResultSet` interface are implemented in `RowSet`.

Note: The Oracle implementation of `ResultSet` provides the scrollable and updatable properties of the `java.sql.ResultSet` interface.

The following code illustrates how to scroll through a RowSet:

```
/**
 *  Scrolling forward, and printing the empno in
 *  the order in which it was fetched.
 */
...
rowset.setCommand("SELECT empno, ename, sal FROM emp");
rowset.execute();
...
// going to the first row of the rowset
rowset.beforeFirst ();
while (rowset.next ())
  System.out.println ("empno: " +rowset.getInt (1));
```

In the preceding code, the cursor position is initialized to the position before the first row of the RowSet by the `beforeFirst` method. The rows are retrieved in forward direction using the `next` method.

The following code illustrates how to scroll through a RowSet in the reverse direction:

```
/**
 *  Scrolling backward, and printing the empno in
 *  the reverse order as it was fetched.
 */
//going to the last row of the rowset
rowset.afterLast ();
while (rowset.previous ())
  System.out.println ("empno: " +rowset.getInt (1));
```

In the preceding code, the cursor position is initialized to the position after the last row of the `RowSet`. The rows are retrieved in reverse direction using the `previous` method of `RowSet`.

Inserting, updating, and deleting rows are supported by the Row Set feature as they are in the Result Set feature. In order to make the Row Set updatable, you must call the `setReadOnly(false)` and `acceptChanges` methods.

The following code illustrates the insertion of a row at the fifth position of a Row Set:

```
...
/**
  * Make rowset updatable
  */
rowset.setReadOnly (false);
/**
 * Inserting a row in the 5th position of the rowset.
 */
// moving the cursor to the 5th position in the rowset
if (rowset.absolute(5))
{
  rowset.moveToInsertRow ();
  rowset.updateInt (1, 193);
  rowset.updateString (2, "Smith");
  rowset.updateInt (3, 7200);
```

```
  // inserting a row in the rowset
  rowset.insertRow ();

  // Synchronizing the data in RowSet with that in the database.
  rowset.acceptChanges ();
}
...
```

In the preceding code, a call to the `absolute` method with a parameter `5` takes the cursor to the fifth position of the RowSet and a call to the `moveToInsertRow` method creates a place for the insertion of a new row into the RowSet. The `update`*`XXX`* methods are used to update the newly created row. When all the columns of the row are updated, the `insertRow` is called to update the RowSet. The changes are committed through `acceptChanges` method.

CachedRowSet

A CachedRowSet is a RowSet in which the rows are cached and the RowSet is disconnected, that is, it does not maintain an active connection to the database. The `oracle.jdbc.rowset.OracleCachedRowSet` class is the Oracle implementation of CachedRowSet. It can interoperate with the standard reference implementation. The `OracleCachedRowSet` class in the `ojdbc6.jar` and `ojdbc7.jar` files implements the standard JSR-114 interface `javax.sql.rowset.CachedRowSet`.

In the following code, an `OracleCachedRowSet` object is created and the connection URL, user name, password, and the SQL query for the `RowSet` object is set as properties. The `RowSet` object is populated using the `execute` method. After the `execute` method has been processed, the `RowSet` object can be used as a `java.sql.ResultSet` object to retrieve, scroll, insert, delete, or update data.

```
...
RowSet rowset = new OracleCachedRowSet();
rowset.setUrl("jdbc:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT employee_id, first_name, last_name, salary FROM
employees");
rowset.execute();
while (rowset.next ())
{
  System.out.println("employee_id: " +rowset.getInt (1));
  System.out.println("first_name: " +rowset.getString (2));
  System.out.println("last_name: " +rowset.getString (3));
  System.out.println("sal: "   +rowset.getInt (4));
}
...
```

To populate a `CachedRowSet` object with a query, complete the following steps:

1. Instantiate `OracleCachedRowSet`.
2. Set the `Url`, which is the connection URL, `Username`, `Password`, and `Command`, which is the query string, properties for the `RowSet` object. You can also set the connection type, but it is optional.
3. Call the `execute` method to populate the `CachedRowSet` object. Calling `execute` runs the query set as a property on this RowSet.

   ```
   OracleCachedRowSet rowset = new OracleCachedRowSet ();
   rowset.setUrl ("jdbc:oracle:oci:@");
   ```

```
    rowset.setUsername ("HR");
    rowset.setPassword ("hr");
    rowset.setCommand ("SELECT employee_id, first_name, last_name, salary FROM
employees");
    rowset.execute ();
```

A `CachedRowSet` object can be populated with an existing `ResultSet` object, using the `populate` method. To do so, complete the following steps:

1. Instantiate `OracleCachedRowSet`.
2. Pass the already available `ResultSet` object to the `populate` method to populate the `RowSet` object.

```
// Executing a query to get the ResultSet object.
ResultSet rset = pstmt.executeQuery ();

OracleCachedRowSet rowset = new OracleCachedRowSet ();
// the obtained ResultSet object is passed to the populate method
// to populate the data in the rowset object.
rowset.populate (rset);
```

In the preceding example, a `ResultSet` object is obtained by running a query and the retrieved `ResultSet` object is passed to the `populate` method of the `CachedRowSet` object to populate the contents of the result set into the CachedRowSet.

> **Note:** Connection properties, like transaction isolation or the concurrency mode of the result set, and the bind properties cannot be set in the case where a pre-existent `ResultSet` object is used to populate the `CachedRowSet` object, because the connection or result set on which the property applies would have already been created.

The following code illustrates how an `OracleCachedRowSet` object is serialized to a file and then retrieved:

```
// writing the serialized OracleCachedRowSet object
{
  FileOutputStream fileOutputStream = new FileOutputStream("emp_tab.dmp");
  ObjectOutputStream ostream = new ObjectOutputStream(fileOutputStream);
  ostream.writeObject(rowset);
  ostream.close();
  fileOutputStream.close();
}

// reading the serialized OracleCachedRowSet object
{
  FileInputStream fileInputStream = new FileInputStream("emp_tab.dmp");
  ObjectInputStream istream = new ObjectInputStream(fileInputStream);
  RowSet rowset1 = (RowSet) istream.readObject();
  istream.close();
  fileInputStream.close();
}
```

In the preceding code, a `FileOutputStream` object is opened for an `emp_tab.dmp` file, and the populated `OracleCachedRowSet` object is written to the file using `ObjectOutputStream`. The serialized `OracleCachedRowSet` object is retrieved using the `FileInputStream` and `ObjectInputStream` objects.

OracleCachedRowSet takes care of the serialization of non-serializable form of data like InputStream, OutputStream, binary large objects (BLOBs), and character large objects (CLOBs). OracleCachedRowSets also implements metadata of its own, which could be obtained without any extra server round-trip. The following code illustrates how you can obtain metadata for the RowSet:

```
...
ResultSetMetaData metaData = rowset.getMetaData();
int maxCol = metaData.getColumnCount();
for (int i = 1; i <= maxCol; ++i)
   System.out.println("Column (" + i +") " + metaData.getColumnName(i));
...
```

Because the OracleCachedRowSet class is serializable, it can be passed across a network or between Java Virtual Machines (JVMs), as done in Remote Method Invocation (RMI). Once the OracleCachedRowSet class is populated, it can move around any JVM, or any environment that does not have JDBC drivers. Committing the data in the RowSet requires the presence of JDBC drivers.

The complete process of retrieving the data and populating it in the OracleCachedRowSet class is performed on the server and the populated RowSet is passed on to the client using suitable architectures like RMI or Enterprise Java Beans (EJB). The client would be able to perform all the operations like retrieving, scrolling, inserting, updating, and deleting on the RowSet without any connection to the database. Whenever data is committed to the database, the acceptChanges method is called, which synchronizes the data in the RowSet to that in the database. This method makes use of JDBC drivers, which require the JVM environment to contain JDBC implementation. This architecture would be suitable for systems involving a Thin client like a Personal Digital Assistant (PDA).

After populating the CachedRowSet object, it can be used as a ResultSet object or any other object, which can be passed over the network using RMI or any other suitable architecture.

Some of the other key-features of CachedRowSet are the following:

- Cloning a RowSet
- Creating a copy of a RowSet
- Creating a shared copy of a RowSet

CachedRowSet Constraints

All the constraints that apply to an updatable result set are applicable here, except serialization, because OracleCachedRowSet is serializable. The SQL query has the following constraints:

- References only a single table in the database
- Contains no join operations
- Selects the primary key of the table it references

In addition, a SQL query should also satisfy the following conditions, if new rows are to be inserted:

- Selects all non-nullable columns in the underlying table
- Selects all columns that do not have a default value

> **Note:** The `CachedRowSet` cannot hold a large quantity of data, because all the data is cached in memory. Oracle, therefore, recommends against using `OracleCachedRowSet` with queries that could potentially return a large volume of data.

Connection properties like, transaction isolation and concurrency mode of the result set, cannot be set after populating the RowSet, because the properties cannot be applied to the connection after retrieving the data from the same.

JdbcRowSet

A JdbcRowSet is a RowSet that wraps around a `ResultSet` object. It is a connected RowSet that provides JDBC interfaces in the form of a JavaBean interface. The Oracle implementation of JdbcRowSet is `oracle.jdbc.rowset.OracleJDBCRowSet`. The `OracleJDBCRowSet` class in `ojdbc6.jar` and `ojdbc7.jar` implements the standard JSR-114 interface `javax.sql.rowset.JdbcRowSet`.

Table 18–1 shows how the `JdbcRowSet` interface differs from `CachedRowSet` interface.

Table 18–1 The JDBC and Cached Row Sets Compared

RowSet Type	Serializable	Connected to Database	Movable Across JVMs	Synchronization of data to database	Presence of JDBC Drivers
JDBC	Yes	Yes	No	No	Yes
Cached	Yes	No	Yes	Yes	No

JdbcRowSet is a connected RowSet, which has a live connection to the database and all the calls on the JdbcRowSet are percolated to the mapping call in the JDBC connection, statement, or result set. A CachedRowSet does not have any connection to the database open.

JdbcRowSet requires the presence of JDBC drivers unlike a CachedRowSet, which does not require JDBC drivers during manipulation. However, both JdbcRowSet and CachedRowSet require JDBC drivers during population of the RowSet and while committing the changes of the RowSet.

The following code illustrates how a JdbcRowSet is used:

```
...
RowSet rowset = new OracleJDBCRowSet();
rowset.setUrl("java:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT empno, ename, sal FROM emp");
rowset.execute();
while (rowset.next())
{
  System.out.println("empno: " + rowset.getInt(1));
  System.out.println("ename: " + rowset.getString(2));
  System.out.println("sal: " + rowset.getInt(3));
}
...
```

In the preceding example, the connection URL, user name, password, and SQL query are set as properties of the `RowSet` object, the SQL query is processed using the

execute method, and the rows are retrieved and printed by traversing through the data populated in the RowSet object.

WebRowSet

A WebRowSet is an extension to CachedRowSet. It represents a set of fetched rows or tabular data that can be passed between tiers and components in a way such that no active connections with the data source need to be maintained. The WebRowSet interface provides support for the production and consumption of result sets and their synchronization with the data source, both in Extensible Markup Language (XML) format and in disconnected fashion. This allows result sets to be shipped across tiers and over Internet protocols.

The Oracle implementation of WebRowSet is oracle.jdbc.rowset.OracleWebRowSet. This class, which is in the ojdbc6.jar and ojdbc7.jar files, implements the standard JSR-114 interface javax.sql.rowset.WebRowSet. This class also extends the oracle.jdbc.rowset.OracleCachedRowSet class. Besides the methods available in OracleCachedRowSet, the OracleWebRowSet class provides the following methods:

```
public OracleWebRowSet() throws SQLException
```

This is the constructor for creating an OracleWebRowSet object, which is initialized with the default values for an OracleCachedRowSet object, a default OracleWebRowSetXmlReader, and a default OracleWebRowSetXmlWriter.

```
public void writeXml(java.io.Writer writer) throws SQLException
public void writeXml(java.io.OutputStream ostream) throws SQLException
```

These methods write the OracleWebRowSet object to the supplied Writer or OutputStream object in the XML format that conforms to the JSR-114 XML schema. In addition to the RowSet data, the properties and metadata of the RowSet are written.

```
public void writeXml(ResultSet rset, java.io.Writer writer) throws SQLException
public void writeXml(ResultSet rset, java.io.OutputStream ostream) throws
SQLException
```

These methods create an OracleWebRowSet object, populate it with the data in the given ResultSet object, and write it to the supplied Writer or OutputStream object in the XML format that conforms to the JSR-114 XML schema.

```
public void readXml(java.io.Reader reader) throws SQLException
public void readXml(java.io.InputStream istream) throws SQLException
```

These methods read the OracleWebRowSet object in the XML format according to its JSR-114 XML schema, using the supplied Reader or InsputStream object.

The Oracle WebRowSet implementation supports Java API for XML Processing (JAXP) 1.2. Both Simple API for XML (SAX) 2.0 and Document Object Model (DOM) JAXP-conforming XML parsers are supported. It follows the current JSR-114 W3C XML schema for WebRowSet at: http://java.sun.com/xml/ns/jdbc/webrowset.xsd

See Also:

- Java SE 6 Javadoc at: http://docs.oracle.com/javase/6/docs/api/
- Java SE 7 Javadoc at: http://docs.oracle.com/javase/7/docs/api/

Applications that use the `readXml(...)` methods should set one of the following two standard JAXP system properties before calling the methods:

- `javax.xml.parsers.SAXParserFactory`

  This property is for a SAX parser.

- `javax.xml.parsers.DocumentBuilderFactory`

  This property is for a DOM parser.

The following code illustrates the use of `OracleWebRowSet` for both writing and reading in XML format:

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.rowset.*;

...
String url = "jdbc:oracle:oci8:@";

Connection conn = DriverManager.getConnection(url,"HR","hr");
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery("select * from employees");

// Create an OracleWebRowSet object and populate it with the ResultSet object
OracleWebRowSet wset = new OracleWebRowSet();
wset.populate(rset);

try
{
  // Create a java.io.Writer object
  FileWriter out = new FileWriter("xml.out");

  // Now generate the XML and write it out
  wset.writeXml(out);
}
catch (IOException exc)
{
  System.out.println("Couldn't construct a FileWriter");
}
System.out.println("XML output file generated.");

// Create a new OracleWebRowSet for reading from XML input
OracleWebRowSet wset2 = new OracleWebRowSet();

// Use Oracle JAXP SAX parser
System.setProperty("javax.xml.parsers.SAXParserFactory","oracle.xml.jaxp.JXSAXPars
erFactory");

try
{
  // Use the preceding output file as input
  FileReader fr = new FileReader("xml.out");

  // Now read XML stream from the FileReader
  wset2.readXml(fr);
}
catch (IOException exc)
{
  System.out.println("Couldn't construct a FileReader");
}
```

```
...
```

Note: The preceding code uses the Oracle SAX XML parser, which supports schema validation.

FilteredRowSet

A FilteredRowSet is an extension to WebRowSet that provides programmatic support for filtering its content. This enables you to avoid the overhead of supplying a query and the processing involved. The Oracle implementation of FilteredRowSet is `oracle.jdbc.rowset.OracleFilteredRowSet`. The `OracleFilteredRowSet` class in the `ojdbc7.jar` files implements the standard JSR-114 interface `javax.sql.rowset.FilteredRowSet`.

The `OracleFilteredRowSet` class defines the following new methods:

```
public Predicate getFilter();
```

This method returns a `Predicate` object that defines the filtering criteria active on the `OracleFilteredRowSet` object.

```
public void setFilter(Predicate p) throws SQLException;
```

This method takes a `Predicate` object as a parameter. The `Predicate` object defines the filtering criteria to be applied on the `OracleFilteredRowSet` object. The methods throws a `SQLException` exception.

The predicate set on an `OracleFilteredRowSet` object defines a filtering criteria that is applied to all the rows in the object to obtain the set of visible rows. The predicate also defines the criteria for inserting, deleting, and modifying rows. The set filtering criteria acts as a gating mechanism for all views and updates to the `OracleFilteredRowSet` object. Any attempt to update the `OracleFilteredRowSet` object, which violates the filtering criteria, throws a `SQLException` exception.

The filtering criteria set on an `OracleFilteredRowSet` object can be modified by applying a new `Predicate` object. The new criteria is immediately applied on the object, and all further views and updates must adhere to this new criteria. A new filtering criteria can be applied only if there are no reference to the `OracleFilteredRowSet` object.

Rows that fall outside of the filtering criteria set on the object cannot be modified until the filtering criteria is removed or a new filtering criteria is applied. Also, only the rows that fall within the bounds of the filtering criteria will be synchronized with the data source, if an attempt is made to persist the object.

The following code example illustrates the use of OracleFilteredRowSet. Assume a table, `test_table`, with two `NUMBER` columns, `col1` and `col2`. The code retrieves those rows from the table that have value of `col1` between `50` and `100` and value of `col2` between `100` and `200`.

The predicate defining the filtering criteria is as follows:

```
public class PredicateImpl implements Predicate
{
  private int low[];
  private int high[];
  private int columnIndexes[];

  public PredicateImpl(int[] lo, int[] hi, int[] indexes)
```

```
  {
    low = lo;
    high = hi;
    columnIndexes = indexes;
  }

  public boolean evaluate(RowSet rs)
  {
    boolean result = true;
    for (int i = 0; i < columnIndexes.length; i++)
    {
      int columnValue = rs.getInt(columnIndexes[i]);
      if (columnValue < low[i] || columnValue > high[i])
        result = false;
    }
    return result;
  }

// the other two evaluate(...) methods simply return true

}
```

The predicate defined in the preceding code is used for filtering content in an `OracleFilteredRowSet` object, as follows:

```
...
OracleFilteredRowSet ofrs = new OracleFilteredRowSet();
int low[] = {50, 100};
int high[] = {100, 200};
int indexes[] = {1, 2};
ofrs.setCommand("select col1, col2 from test_table");

// set other properties on ofrs like usr/pwd ...
...
ofrs.execute();
ofrs.setPredicate(new PredicateImpl(low, high, indexes));

// this will only get rows with col1 in (50,100) and col2 in (100,200)
while (ofrs.next()) {...}
...
```

JoinRowSet

A JoinRowSet is an extension to WebRowSet that consists of related data from different RowSets. There is no standard way to establish a SQL `JOIN` between disconnected RowSets without connecting to the data source. A JoinRowSet addresses this issue. The Oracle implementation of JoinRowSet is the `oracle.jdbc.rowset.OracleJoinRowSet` class. This class, which is in the `ojdbc7.jar` files, implements the standard JSR-114 interface `javax.sql.rowset.JoinRowSet`.

Any number of `RowSet` objects, which implement the `Joinable` interface, can be added to a `JoinRowSet` object, provided they can be related in a SQL `JOIN`. All five types of RowSet support the `Joinable` interface. The `Joinable` interface provides methods for specifying the columns based on which the `JOIN` will be performed, that is, the match columns.

A match column can be specified in the following ways:

- Using the `setMatchColumn` method

This method is defined in the `Joinable` interface. It is the only method that can be used to set the match column before a `RowSet` object is added to a `JoinRowSet` object. This method can also be used to reset the match column at any time.

- Using the `addRowSet` method

  This is an overloaded method in `JoinRowSet`. Four of the five implementations of this method take a match column as a parameter. These four methods can be used to set or reset a match column at the time a `RowSet` object is being added to a `JoinRowSet` object.

In addition to the inherited methods, `OracleJoinRowSet` provides the following methods:

```
public void addRowSet(Joinable joinable) throws SQLException;
public void addRowSet(RowSet rowSet, int i) throws SQLException;
public void addRowSet(RowSet rowSet, String s) throws SQLException;
public void addRowSet(RowSet arowSet[], int an[]) throws SQLException;
public void addRowSet(RowSet arowSet[], String as[]) throws SQLException;
```

These methods are used to add a `RowSet` object to the `OracleJoinRowSet` object. You can pass one or more `RowSet` objects to be added to the `OracleJoinRowSet` object. You can also pass names or indexes of one or more columns, which need to be set as match column.

```
public Collection getRowSets() throws SQLException;
```

This method retrieves the `RowSet` objects added to the `OracleJoinRowSet` object. The method returns a `java.util.Collection` object that contains the `RowSet` objects.

```
public String[] getRowSetNames() throws SQLException;
```

This method returns a String array containing the names of the `RowSet` objects that are added to the `OracleJoinRowSet` object.

```
public boolean supportsCrossJoin();
public boolean supportsFullJoin();
public boolean supportsInnerJoin();
public boolean supportsLeftOuterJoin();
public boolean supportsRightOuterJoin();
```

These methods return a boolean value indicating whether the `OracleJoinRowSet` object supports the corresponding `JOIN` type.

```
public void setJoinType(int i) throws SQLException;
```

This method is used to set the `JOIN` type on the `OracleJoinRowSet` object. It takes an integer constant as defined in the `javax.sql.rowset.JoinRowSet` interface that specifies the `JOIN` type.

```
public int getJoinType() throws SQLException;
```

This method returns an integer value that indicates the `JOIN` type set on the `OracleJoinRowSet` object. This method throws a `SQLException` exception.

```
public CachedRowSet toCachedRowSet() throws SQLException;
```

This method creates a `CachedRowSet` object containing the data in the `OracleJoinRowSet` object.

```
public String getWhereClause() throws SQLException;
```

This method returns a String containing the SQL-like description of the `WHERE` clause used in the `OracleJoinRowSet` object. This methods throws a `SQLException` exception.

The following code illustrates how `OracleJoinRowSet` is used to perform an inner join on two RowSets, whose data come from two different tables. The resulting RowSet contains data as if they were the result of an inner join on these two tables. Assume that there are two tables, an `Order` table with two `NUMBER` columns `Order_id` and `Person_id`, and a `Person` table with a `NUMBER` column `Person_id` and a `VARCHAR2` column `Name`.

```
...
// RowSet holding data from table Order
OracleCachedRowSet ocrsOrder = new OracleCachedRowSet();
...
ocrsOrder.setCommand("select order_id, person_id from order");
...
// Join on person_id column
ocrsOrder.setMatchColumn(2);
ocrsOrder.execute();

// Creating the JoinRowSet
OracleJoinRowSet ojrs = new OracleJoinRowSet();
ojrs.addRowSet(ocrsOrder);

// RowSet holding data from table Person
OracleCachedRowSet ocrsPerson = new OracleCachedRowSet();
...
ocrsPerson.setCommand("select person_id, name from person");
...
// do not set match column on this RowSet using setMatchColumn().
//use addRowSet() to set match column
ocrsPerson.execute();

// Join on person_id column, in another way
ojrs.addRowSet(ocrsPerson, 1);

// now we can go the JoinRowSet as usual
ojrs.beforeFirst();
while (ojrs.next())
System.out.println("order id = " + ojrs.getInt(1) + ", " + "person id = " +
ojrs.getInt(2) + ", " + "person's name = " + ojrs.getString(3));
...
```

19
Globalization Support

The Oracle Java Database Connectivity (JDBC) drivers provide globalization support, formerly known as National Language Support (NLS). Globalization support enables you to retrieve data or insert data into a database in any character set that Oracle supports. If the clients and the server use different character sets, then the driver provides the support to perform the conversions between the database character set and the client character set.

This chapter contains the following sections:

See Also:

- "Oracle Character Data Types Support" on page 4-9
- *Oracle Database Globalization Support Guide*
- *Oracle Database Reference*

Note:

- Starting from Oracle Database 10*g*, the `NLS_LANG` variable is no longer part of the JDBC globalization mechanism. The JDBC driver does not check NLS environment. So, setting it has no effect.
- The JDBC server-side internal driver provides complete globalization support and does not require any globalization extension files.
- JDBC 4.0 includes methods for reading and writing national character set values. You should use these methods when using JSE 6 or later.

Providing Globalization Support

The basic Java Archive (JAR) file `ojdbc7.jar`, contains all the necessary classes to provide complete globalization support for:

- Oracle character sets for `CHAR`, `VARCHAR`, `LONGVARCHAR`, or `CLOB` data that is not being retrieved or inserted as a data member of an Oracle object or collection type.

- CHAR or VARCHAR data members of object and collection for the character sets US7ASCII, WE8DEC, WE8ISO8859P1, WE8MSWIN1252, and UTF8.

To use any other character sets in CHAR or VARCHAR data members of objects or collections, you must include orai18n.jar in the CLASSPATH environment variable:

ORACLE_HOME/jlib/orai18n.jar

Note: Previous releases depended on the nls_charset12.zip file. This file is now obsolete.

Compressing orai18n.jar

The orai18n.jar file contains many important character set and globalization support files. You can reduce the size of orai18n.jar using the built-in customization tool, as follows:

```
java -jar orai18n.jar -custom-charsets-jar [jar/zip_filename] -charset
characterset_name [characterset_name ...]
```

For example, if you want to create a custom character set file, custom_orai18n_ja.jar, that includes the JA16SJIS and JA16EUC character sets, then issue the following command:

```
$ java -jar orai18n.jar -custom-charsets-jar custom_orai18n_ja.jar -charset
JA16SJIS JA16EUC
```

The output of the command is as follows:

```
Added Character set : JA16SJIS
Added Character set : JA16EUC
```

If you do not specify a file name for your custom JAR/ZIP file, then a file with the name jdbc_orai18n_cs.jar is created in the current working directory. Also, for your custom JAR/ZIP file, you cannot specify a name that starts with orai18n.

If any invalid or unsupported character set name is specified in the command, then no output JAR/ZIP file will be created. If the custom JAR/ZIP file exists, then the file will not be updated or removed.

The custom character set JAR/ZIP does not accept any command. However, it prints the version information and the command that was used to generate the JAR/ZIP file. For example, you have jdbc_orai18n_cs.zip, the command that displays the information and the displayed information is as follows:

```
$ java -jar jdbc_orai18n_cs.jar
Oracle Globalization Development Kit - 12.1.X.X.X Release
This custom character set jar/zip file was created with the following command:
java -jar orai18n.jar -custom-charsets-jar jdbc_orai18n_cs.jar -charset
WE8ISO8859P15
```

The limitation to the number of character sets that can be specified depends on that of the shell or command prompt of the operating system. It is certified that all supported character sets can be specified with the command.

Note: If you are using a custom character set, then you need to perform the following so that JDBC supports the custom character set:

1. After creating the `.nlt` and `.nlb` files as part of the process of creating a custom character set, create `.glb` files for the newly created character set and also for the `lx0boot.nlt` file using the following command:

   ```
   java -classpath $ORACLE_HOME/jlib/orai18n-tools.jar Ginstall
   <nlt file>
   ```

2. Add the generated files and `$ORACLE_HOME/jlib/orai18n-mappings.jar` into the `classpath` environment variable while executing the JDBC code that connects to the database with the custom character set.

For more information about creating a custom character set, refer to *Oracle Database Globalization Support Guide.*

NCHAR, NVARCHAR2, NCLOB and the defaultNChar Property

By default, the `oracle.jdbc.OraclePreparedStatement` interface treats the data type of all the columns in the same way as they are encoded in the database character set. However, since Oracle Database 10*g*, if you set the value of `oracle.jdbc.defaultNChar` system property to `true`, then JDBC treats all character columns as being national-language.

The default value of `defaultNChar` is false. If the value of `defaultNChar` is false, then you must call the `setFormOfUse(<column_Index>, OraclePreparedStatement.FORM_NCHAR)` method for those columns that specifically need national-language characters. For example:

```
PreparedStatement pstmt =
conn.prepareStatement("insert into TEST values(?,?,?)");
pstmt.setFormOfUse(1, OraclePreparedStatement.FORM_NCHAR);
pstmt.setString(1, myUnicodeString1); // NCHAR column
pstmt.setFormOfUse(2, OraclePreparedeStatement.FORM_NCHAR);
pstmt.setString(2, myUnicodeString2); // NVARCHAR2 column
```

If you want to set the value of `defaultNChar` to `true`, then specify the following at the command-line:

```
java -Doracle.jdbc.defaultNChar=true myApplication
```

If you prefer, then you can also specify `defaultNChar` as a connection property and access `NCHAR`, `NVARCHAR2`, or `NCLOB` data.

```
Properties props = new Properties();
props.put(OracleConnection.CONNECTION_PROPERTY_DEFAULTNCHAR, "true");
// set URL, username, password, and so on.
...
Connection conn = DriverManager.getConnection(props);
```

If the value of `defaultNChar` is `true`, then you should call the `setFormOfUse(<column_Index>, FORM_CHAR)` for columns that do not need national-language characters. For example:

```
PreparedStatement pstmt =
conn.prepareStatement("insert into TEST values(?,?,?)");
pstmt.setFormOfUse(3, OraclePreparedStatement.FORM_CHAR);
pstmt.setString(3, myString); // CHAR column
```

Note:

- Always use `java.lang.String` for character data instead of `oracle.sql.CHAR`. `CHAR` is provided only for backward compatibility.
- You can also use the `setObject` method to access national character set types, but if the `setObject` method is used, then the target data type must be specified as `Types.NCHAR`, `Types.NCLOB`, `Types.NVARCHAR`, or `Types.LONGNVARCHAR`.

Note: In Oracle Database, SQL strings are converted to the database character set. Therefore you need to keep in mind the following:

- In Oracle Database 10*g* release 1 (10.1) and earlier releases, JDBC drivers do not support any `NCHAR` literal (n'...') containing Unicode characters that are not representable in the database character set. All Unicode characters that are not representable in the database character set get corrupted.
- If an Oracle Database 10*g* release 2 (10.2) JDBC driver is connected to an Oracle Database 10*g* release 2 (10.2) database server, then all `NCHAR` literals (n'...') are converted to Unicode literals (u'...') and all non-ASCII characters are converted to their corresponding Unicode escape sequence. This is done automatically to prevent data corruption.
- If an Oracle Database 10*g* release 2 (10.2) JDBC driver is connected to an Oracle Database 10*g* release 1 (10.1) or earlier database server, then `NCHAR` literals (n'...') are not converted and any character that is not representable in the database character set gets corrupted.

New Methods for National Character Set Type Data in JDK 6

JDBC 4.0 introduces support for the following four additional SQL types to access the national character set types:

- `NCHAR`
- `NVARCHAR`
- `LONGNVARCHAR`
- `NCLOB`

These types are similar to the `CHAR`, `VARCHAR`, `LONGVARCHAR`, and `CLOB` types, except that the values are encoded using the national character set. The JDBC specification uses the `String` class to represent `NCHAR`, `NVARCHAR`, and `LONGNVARCHAR` data, and the `NClob` class to represent `NCLOB` values.

To retrieve a national character value, an application calls one of the following methods:

- `getNString`
- `getNClob`
- `getNCharacterStream`

- `getObject`

To specify a value for a parameter marker of national character type, an application calls one of the following methods:

- `setNString`
- `setNCharacterStream`
- `setNClob`
- `setObject`

Note: You can use the `setFormOfUse` method to specify a national character value in JDK 6. But this practice is discouraged because this method will be deprecated in future release. So, Oracle recommends you to use the methods discussed in this section.

See Also: If the `setObject` method is used, then the target data type must be specified as `Types.NCHAR`, `Types.NCLOB`, `Types.NVARCHAR`, or `Types.LONGNVARCHAR`.

Part V

Performance and Scalability

This part consists of chapters that discuss the Oracle Java Database Connectivity (JDBC) features that enhance performance, such as Statement caching and Oracle Call Interface (OCI) connection pooling. It also includes a chapter that provides information about Oracle performance extensions, such as update batching and row prefetching.

Part V contains the following chapters:

20

Statement and Result Set Caching

This chapter describes the benefits and use of Statement caching, an Oracle Java Database Connectivity (JDBC) extension.

> **Note:** Use statement caching only when you are sure that the table structure remains the same in the database. If you alter the table structure and then reuse a statement that was created and executed before changing the table structure, then you may get an error.

This chapter contains the following sections:

About Statement Caching

Statement caching improves performance by caching executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. Starting from JDBC 3.0, JDBC standards define a statement-caching interface.

Statement caching can do the following:

- Prevent the overhead of repeated cursor creation
- Prevent repeated statement parsing and creation
- Reuse data structures in the client

This section covers the following topics:

> **Note:** Oracle strongly recommends you use the implicit Statement cache. Oracle JDBC drivers are designed on the assumption that the implicit Statement cache is enabled. So, not using the Statement cache will have a negative impact on performance.

Basics of Statement Caching

Applications use the Statement cache to cache statements associated with a particular physical connection. The cache is associated with an `OracleConnection` object. `OracleConnection` includes methods to enable Statement caching. When you enable Statement caching, a statement object is cached when you call the `close` method.

Because each physical connection has its own cache, multiple caches can exist if you enable Statement caching for multiple physical connections. When you enable Statement caching on a connection cache, the logical connections benefit from the Statement caching that is enabled on the underlying physical connection. If you try to enable Statement caching on a logical connection held by a connection cache, then this will throw an exception.

There are two types of Statement caching: implicit and explicit. Each type of Statement cache can be enabled or disabled independent of the other. You can have either, neither, or both in effect. Both types of Statement caching share a single cache per connection.

Implicit Statement Caching

When you enable implicit Statement caching, JDBC automatically caches the prepared or callable statement when you call the `close` method of this statement object. The prepared and callable statements are cached and retrieved using standard connection object and statement object methods.

Plain statements are not implicitly cached, because implicit Statement caching uses a SQL string as a key and plain statements are created without a SQL string. Therefore, implicit Statement caching applies only to the `OraclePreparedStatement` and `OracleCallableStatement` objects, which are created with a SQL string. You *cannot* use implicit Statement caching with `OracleStatement`. When you create an `OraclePreparedStatement` or `OracleCallableStatement`, the JDBC driver automatically searches the cache for a matching statement. The match criteria are the following:

- The SQL string in the statement must be identical to one in the cache.
- The statement type must be the same, that is, prepared or callable.
- The scrollable type of result sets produced by the statement must be the same, that is, forward-only or scrollable.

If a match is found during the cache search, then the cached statement is returned. If a match is not found, then a new statement is created and returned. In either case, the statement, along with its cursor and state are cached when you call the `close` method of the statement object.

When a cached `OraclePreparedStatement` or `OracleCallableStatement` object is retrieved, the state and data information are automatically reinitialized and reset to default values, while metadata is saved. Statements are removed from the cache to conform to the maximum size using a Least Recently Used (LRU) algorithm.

Note: The JDBC driver does not clear metadata. However, although metadata is saved for performance reasons, it has no semantic impact. A statement that comes from the implicit cache appears as if it were newly created.

You can prevent a particular statement from being implicitly cached.

See Also: "Disabling Implicit Statement Caching for a Particular Statement" on page 20-6

Explicit Statement Caching

Explicit Statement caching enables you to cache and retrieve selected prepared and callable statements. Explicit Statement caching relies on a key, an arbitrary Java `String` that you provide.

> **Note:** Plain statements cannot be cached.

Because explicit Statement caching retains statement data and state as well as metadata, it has a performance edge over implicit Statement caching, which retains only metadata. However, you must be cautious when using this type of caching, because explicit Statement caching saves all three types of information for reuse and you may not be aware of what data and state are retained from prior use of the statements.

Implicit and explicit Statement caching can be differentiated on the following points:

- Retrieving statements

  In the case of implicit Statement caching, you take no special action to retrieve statements from a cache. Instead, whenever you call `prepareStatement` or `prepareCall`, JDBC automatically checks the cache for a matching statement and returns it if found. However, in the case of explicit Statement caching, you use specialized Oracle `WithKey` methods to cache and retrieve statement objects.

- Providing key

  Implicit Statement caching uses the SQL string of a prepared or callable statement as the key, requiring no action on your part. In contrast, explicit Statement caching requires you to provide a Java `String`, which it uses as the key.

- Returning statements

  During implicit Statement caching, if the JDBC driver cannot find a statement in cache, then it will automatically create one. However, during explicit Statement caching, if the JDBC driver cannot find a matching statement in cache, then it will return a `null` value.

Table 20–1 compares the different methods employed in implicit and explicit Statement caching.

Table 20–1 Comparing Methods Used in Statement Caching

	Allocate	Insert Into Cache	Retrieve From Cache
Implicit	`prepareStatement` `prepareCall`	`close`	`prepareStatement` `prepareCall`

Explicit	`createStatement`
	`prepareStatement`
	`prepareCall`
		`closeWithKey`	`getStatementWithKey`
			`getCallWithKey`

Using Statement Caching

This section discusses the following topics:

- Enabling and Disabling Statement Caching

Enabling and Disabling Statement Caching

When using the `OracleConnection` API, implicit and explicit Statement caching can be enabled or disabled independent of one other. You can have either, neither, or both of them in effect.

Enabling Implicit Statement Caching

There are two ways to enable implicit Statement caching. The first method enables Statement caching on a nonpooled physical connection, where you need to explicitly specify the Statement size for every connection, using the `setStatementCacheSize` method. The second method enables Statement caching on a pooled logical connection. Each connection in the pool has its own Statement cache with the same maximum size that can be specified by setting the `MaxStatementsLimit` property.

Method 1

Perform the following steps:

- Call the `OracleDataSource.setImplicitCachingEnabled(true)` method on the connection to set the `OracleDataSource` property `implicitCachingEnabled` to `true`. For example:

```
OracleDataSource ods =  new OracleDataSource();
...
ods.setImplicitCachingEnabled(true);
...
```

- Call the `OracleConnection.setStatementCacheSize` method on the physical connection. The argument you supply is the maximum number of statements in the cache. For example, the following code specifies a cache size of ten statements:

```
((OracleConnection)conn).setStatementCacheSize(10);
```

Method 2

Perform the following steps:

- Set the `OracleDataSource` properties `implicitCachingEnabled` and `connectionCachingEnabled` to `true`. For example:

```
OracleDataSource ods =  new OracleDataSource();
...
ods.setConnectionCachingEnabled( true );
ods.setImplicitCachingEnabled( true );
...
```

- Set the `MaxStatementsLimit` property to a positive integer on the connection cache, when using the connection cache. For example:

```
Properties cacheProps = new Properties();
...
cacheProps.put( "MaxStatementsLimit", "50" );
```

To determine whether implicit caching is enabled, call `getImplicitCachingEnabled`, which returns `true` if implicit caching is enabled, `false` otherwise.

Note: Enabling Statement caching enables both implicit and explicit Statement caching.

Disabling Implicit Statement Caching

Disable implicit Statement caching by calling `setImplicitCachingEnabled(false)` on the connection or by setting the `ImplicitCachingEnabled` property to `false`.

Enabling Explicit Statement Caching

To enable explicit Statement caching you must first set the Statement cache size. For setting the cache size, call `OracleConnection.setStatementCacheSize` method on the physical connection. The argument you supply is the maximum number of statements in the cache. An argument of `0` specifies no caching. To check the cache size, use the `getStatementCacheSize` method in the following way:

```
System.out.println("Stmt Cache size is " +
   ((OracleConnection)conn).getStatementCacheSize());
```

The following code specifies a cache size of ten statements:

```
((OracleConnection)conn).setStatementCacheSize(10);
```

Enable explicit Statement caching by calling `setExplicitCachingEnabled(true)` on the connection.

To determine whether explicit caching is enabled, call `getExplicitCachingEnabled`, which returns `true` if explicit caching is enabled, `false` otherwise.

Note:

- You enable implicit and explicit caching for a particular physical connection independently. Therefore, it is possible to do Statement caching both implicitly and explicitly during the same session.
- Implicit and explicit Statement caching share the *same* cache. Remember this when you set the statement cache size.

Disabling Explicit Statement Caching

Disable explicit Statement caching by calling `setExplicitCachingEnabled(false)`. Disabling caching or closing the cache purges the cache. The following example disables explicit Statement caching:

```
((OracleConnection)conn).setExplicitCachingEnabled(false);
```

Closing a Cached Statement

Perform the following to close a Statement and assure that it is not returned to the cache:

In J2SE 5.0

- Disable caching for that statement

  ```
  stmt.setDisableStmtCaching(true);
  ```

- Call the `close` method of the statement object

```
stmt.close();
```

In JSE 6.0

```
stmt.setPoolable(false);
stmt.close();
```

Physically Closing a Cached Statement

With implicit Statement caching enabled, you cannot physically close statements manually. The `close` method of a statement object caches the statement instead of closing it. The statement is physically closed automatically under one of following three conditions:

- When the associated connection is closed
- When the cache reaches its size limit and the least recently used statement object is preempted from cache by the LRU algorithm
- If you call the `close` method on a statement for which Statement caching is disabled

Using Implicit Statement Caching

Once you enable implicit Statement caching, by default, all prepared and callable statements are automatically cached. Implicit Statement caching includes the following steps:

1. Enable implicit Statement caching.
2. Allocate a statement using one of the standard methods.
3. Disable implicit Statement caching for any particular statement you do not want to cache. This is an optional step.
4. Cache the statement using the `close` method.
5. Retrieve the implicitly cached statement by calling the appropriate standard prepare method.

Allocating a Statement for Implicit Caching

To allocate a statement for implicit Statement caching, use either the `prepareStatement` or `prepareCall` method as you would typically.

The following code allocates a new statement object called `pstmt`:

```
PreparedStatement pstmt = conn.prepareStatement
   ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

Disabling Implicit Statement Caching for a Particular Statement

With implicit Statement caching enabled for a connection, by default, all callable and prepared statements of that connection are automatically cached. To prevent a particular callable or prepared statement from being implicitly cached, use the `setDisableStmtCaching` method of the statement object. You can manage cache space by calling the `setDisableStmtCaching` method on any infrequently used statement.

The following code disables implicit Statement caching for `pstmt`:

```
PreparedStatement pstmt = conn.prepareStatement("SELECT 1 from DUAL");
((OraclePreparedStatement)pstmt).setDisableStmtCaching(true);
pstmt.close ();
```

Note: If you are using JSE 6, then you can disable Statement caching by using the standard JDBC 4.0 method `setPoolable`:

```
PreparedStatement.setPoolable(false);
```

Use the following to check whether the `Statement` object is poolable:

```
Statement.isPoolable();
```

Implicitly Caching a Statement

To cache an allocated statement, call the `close` method of the statement object. When you call the `close` method on an `OraclePreparedStatement` or `OracleCallableStatement` object, the JDBC driver automatically puts this statement in cache, unless you have disabled caching for this statement.

The following code caches the `pstmt` statement:

```
pstmt.close ();
```

Retrieving an Implicitly Cached Statement

To retrieve an implicitly cached statement, call either the `prepareStatement` or `prepareCall` method, depending on the statement type.

The following code retrieves `pstmt` from cache using the `prepareStatement` method:

```
pstmt = conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

Table 20–2 describes the methods used to allocate statements and retrieve implicitly cached statements.

Table 20–2 Methods Used in Statement Allocation and Implicit Statement Caching

Method	Functionality for Implicit Statement Caching
`prepareStatement`	Performs a cache search that either finds and returns the desired cached `OraclePreparedStatement` object or allocates a new `OraclePreparedStatement` object if a match is not found
`prepareCall`	Performs a cache search that either finds and returns the desired cached `OracleCallableStatement` object or allocates a new `OracleCallableStatement` object if a match is not found

Example 20–1 provides a sample code that shows how to enable implicit statement caching.

Example 20–1 Using Implicit Statement Cache

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import javax.sql.DataSource;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
public class TestJdbc
{
  /**
```

```
   * Get a Connection, prepare a statement, execute a query, fetch the results,
close the connection.
   * @param ods the DataSource used to get the connection.
   */
  private static void doSQL( DataSource ods ) throws SQLException
  {
    final String SQL = "select username from all_users";
    OracleConnection  conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try
    {
      conn = (OracleConnection) ods.getConnection();
      System.out.println( "Connection:" + conn );
      System.out.println( "Connection getImplicitCachingEnabled:" +
conn.getImplicitCachingEnabled() );
      System.out.println( "Connection getStatementCacheSize:" +
conn.getStatementCacheSize() );
      ps = conn.prepareStatement( SQL );
      System.out.println( "PreparedStatement:" + ps );
      rs = ps.executeQuery();
      while ( rs.next() )
      {
        String owner = rs.getString( 1 );
        System.out.println( owner );
      }
    }
    finally
    {
      if ( rs != null )
      {
        rs.close();
      }
      if ( ps != null )
      {
        ps.close();
      conn.close();
    }
  }
  }
  public static void main( String[] args )
  {
    try
    {
      OracleDataSource ods =  new OracleDataSource();
      ods.setDriverType( "thin" );
      ods.setServerName( "localhost" );
      ods.setPortNumber( 5221 );
      ods.setServiceName( "orcl" );
      ods.setUser( "HR" );
      ods.setPassword( "hr" );
      ods.setConnectionCachingEnabled( true );
      ods.setImplicitCachingEnabled( true );
      Properties cacheProps = new Properties();
      cacheProps.put( "InitialLimit", "1" );
      cacheProps.put( "MinLimit", "1" );
      cacheProps.put( "MaxLimit", "5" );
      cacheProps.put( "MaxStatementsLimit", "50" );
      ods.setConnectionCacheProperties( cacheProps );
      System.out.println( "DataSource getImplicitCachingEnabled: " +
```

```
ods.getImplicitCachingEnabled() );
      for ( int i = 0; i < 5; i++ )
      {
        doSQL( ods );
      }
    }
    catch ( Exception ex )
    {
      ex.printStackTrace();
    }
  }
}
```

Using Explicit Statement Caching

A prepared or callable statement can be explicitly cached when you enable explicit Statement caching. Explicit Statement caching includes the following steps:

1. Enable explicit Statement caching.
2. Allocate a statement using one of the standard methods.
3. Explicitly cache the statement by closing it with a key, using the `closeWithKey` method.
4. Retrieve the explicitly cached statement by calling the appropriate Oracle WithKey method, specifying the appropriate key.
5. Re-cache an open, explicitly cached statement by closing it again with the `closeWithKey` method. Each time a cached statement is closed, it is re-cached with its key.

Allocating a Statement for Explicit Caching

To allocate a statement for explicit Statement caching, use either the `createStatement`, `prepareStatement`, or `prepareCall` method as you would typically.

The following code allocates a new statement object called `pstmt`:

```
PreparedStatement pstmt =
   conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

Explicitly Caching a Statement

To explicitly cache an allocated statement, call the `closeWithKey` method of the statement object, specifying a key. The key is an arbitrary Java `String` that you provide. The `closeWithKey` method caches a statement as is. This means the data, state, and metadata are retained and not cleared.

The following code caches the `pstmt` statement with the key `"mykey"`:

```
((OraclePreparedStatement)pstmt).closeWithKey ("mykey");
```

Retrieving an Explicitly Cached Statement

To recall an explicitly cached statement, call either the `getStatementWithKey` or `getCallWithKey` methods depending on the statement type.

If you retrieve a statement with a specified key, then the JDBC driver searches the cache for the statement, based on the specified key. If a match is found, then the matching statement is returned along with its state, data, and metadata. This information is as it was when the statement was last closed. If a match is not found, then the JDBC driver returns `null`.

The following code recalls `pstmt` from cache using the `"mykey"` key with the `getStatementWithKey` method. Recall that the `pstmt` statement object was cached with the `"mykey"` key.

```
pstmt = ((OracleConnection)conn).getStatementWithKey ("mykey");
```

If you call the `creationState` method on the `pstmt` statement object, then the method returns `EXPLICIT`.

Important: When you retrieve an explicitly cached statement, ensure that you use the method that is appropriate for your statement type when specifying the key. For example, if you used the `prepareStatement` method to allocate a statement, then use the `getStatementWithKey` method to retrieve that statement from cache. The JDBC driver does *not* verify the type of statement it is returning.

Table 20–3 describes the methods used to retrieve explicitly cached statements.

Table 20–3 Methods Used to Retrieve Explicitly Cached Statements

Method	Functionality for Explicit Statement Caching
`getStatementWithKey`	Specifies the key needed to retrieve a prepared statement from cache
`getCallWithKey`	Specifies the key needed to retrieve a callable statement from cache

Reusing Statements Objects

The JDBC 3.0 specification introduces the feature of statement pooling that enables an application to reuse a `PreparedStatement` object in the same way as it uses a `Connection` object. The `PreparedStatement` objects can be reused by multiple logical connections in a transparent manner.

This section covers the following topics:

- Using a Pooled Statement
- Closing a Pooled Statement

Note: The Oracle JDBC Drivers use implicit statement caching to support statement pooling.

Using a Pooled Statement

An application can find out whether a data source supports statement pooling by calling the `isPoolable` method from the `Statement` interface. If the return value is `true`, then the application knows that the `PreparedStatement` object is being pooled. The application can also request a statement to be pooled or not pooled by using the `setPoolable` method from the `Statement` interface.

Reusing of pooled statement should be completely transparent to the application, that is, the application code should remain the same whether a `PreparedStatement` object participates in statement pooling or not. If an application closes a `PreparedStatement` object, it must still call `Connection.prepareStatement` method in order to reuse it.

Note: An application has no direct control over how statements are pooled. A pool of statements is associated with a `PooledConnection` object, whose behavior is determined by the properties of the `ConnectionPoolDataSource` object that produced it.

Closing a Pooled Statement

An application closes a pooled statement exactly the same way it closes a nonpooled statement. Once a statement is closed, whether is it pooled or nonpooled, it is no longer available for use by the application and an attempt to reuse it causes an exception to be thrown. The only difference visible is that an application cannot directly close a physical statement that is being pooled. This is done by the pool manager. The method `PooledConnection.closeAll` closes all of the statements open on a given physical connection, which releases the resources associated with those statements.

The following methods can close a pooled statement:

- `close`

  This `java.sql.Statement` interface method is called by an application. If the statement is being pooled, then it closes the logical statement used by the application but does not close the physical statement being pooled.

- `close`

  This `java.sql.Connection` interface method is called by an application. This method acts differently depending upon whether the connection using the statement is being pooled or not:

  – Nonpooled connection

    This method closes the physical connection and all statements created by that connection. This is necessary because the garbage collection mechanism is unable to detect when externally managed resources can be released.

  – Pooled connection

    This method closes the logical connection and the logical statements it returned, but leaves open the underlying `PooledConnection` object and any associated pooled statements

- `PooledConnection.closeAll`

  This method is called by the connection pool manager to close all of the physical statements being pooled by the `PooledConnection` object

Result Set Caching

Your applications sometime send repetitive queries to the database. To improve the response time of repetitive queries, results of queries, query fragments, and PL/SQL functions can be cached in memory. A result cache stores the results of queries shared across all sessions. When these queries are executed repeatedly, the results are retrieved directly from the cache memory.

You must annotate a query or query fragment with a result cache hint to indicate that results are to be stored in the query result cache.

The query result set can be cached in the following ways:

- Server-side Cache

- Client Result Cache

Note:

- The server-side and client result set caches are most useful for read-only or read-mostly data. They may reduce performance for queries with highly dynamic results.
- Both server-side and client result set caches use memory. So, caching very large result sets can cause performance problems.

Server-side Cache

Support for server-side Result Set caching has been introduced for both JDBC Thin and JDBC Oracle Call Interface (OCI) drivers since Oracle Database 11*g* Release 1. The server-side result cache is used to cache the results of the current queries, query fragments, and PL/SQL functions in memory and then to use the cached results in future executions of the query, query fragment, or PL/SQL function. The cached results reside in the result cache memory portion of the SGA. A cached result is automatically invalidated whenever a database object used in its creation is successfully modified. The server-side caching can be of the following two types:

- SQL Query Result Cache
- PL/SQL Function Result Cache

See Also:

- *Oracle Database Performance Tuning Guide* for more information about SQL Query Result Cache
- *Oracle Database PL/SQL Language Reference* for more information about PL/SQL Function Result Cache

Client Result Cache

Since Oracle Database 11*g* Release 1, support for client result cache has been introduced for JDBC OCI driver. The client result cache improves performance of applications by caching query result sets in a way that subsequent query executions can access the cached result set without fetching rows from the server. This eliminates many round-trips to the server for cached results and reduces CPU usage on the server. The client cache transparently keeps the result set consistent with any session state or database changes that can affect its cached result sets. This allows significant improvements in response time for frequent client SQL query executions and for fetching rows. The scalability on the server is increased since it expends less CPU time.

See Also: Client Result Cache on page 6-1

21
Performance Extensions

This chapter describes the Oracle performance extensions to the Java Database Connectivity (JDBC) standard.

This chapter covers the following topics:

- Update Batching
- Additional Oracle Performance Extensions

> **Note:** Starting from Oracle Database 12*c* Release 1 (12.1), Oracle update batching is deprecated. Oracle recommends that you use standard JDBC batching instead of Oracle update batching.

Update Batching

You can reduce the number of round-trips to the database, thereby improving application performance, by grouping multiple `UPDATE`, `DELETE`, or `INSERT` statements into a single batch and having the whole batch sent to the database and processed in one trip. This is referred to as 'update batching'.

> **Note:** The JDBC 2.0 specification refers to 'update batching' as 'batch updates'.

This is especially useful with prepared statements, when you are repeating the same statement with different bind variables.

Oracle JDBC supports two distinct models for update batching:

- The standard model, implementing the JDBC 2.0 specification, which is referred to as standard update batching
- The Oracle-specific model, independent of the JDBC 2.0 specification, which is referred to as Oracle update batching

> **Note:** It is important to be aware that you cannot mix these models. In any single application, you can use one model or the other, but not both. Oracle JDBC driver will throw exceptions when you mix these.

This section covers the following topics:

- Overview of Update Batching Models

Overview of Update Batching Models

This section compares and contrasts the general models and types of statements supported for standard update batching and Oracle update batching.

Note: Starting from Oracle Database 12*c* Release 1 (12.1), Oracle update batching is deprecated. Oracle recommends that you use standard JDBC batching instead of Oracle update batching.

Oracle Model Versus Standard Model

Oracle update batching uses a batch value that typically results in implicit processing of a batch. The batch value is the number of operations you want to add to a batch for each trip to the database. As soon as that many operations have been added to the batch, the batch is processed. Note the following:

- You can set a default batch for the connection object, which applies to any prepared statement run in that connection.
- For any individual prepared statement object, you can set a statement batch value that overrides the connection batch value.
- You can choose to explicitly process a batch at any time, overriding both the connection batch value and the statement batch value.

Standard update batching is a manual, explicit model. There is no batch value. You manually add operations to the batch, and then, explicitly choose when to process the batch.

Note:

- Oracle recommends that you use JDBC standard features when possible. This recommendation applies to update batching as well. Oracle update batching is retained primarily for backwards compatibility.
- For both standard update batching and Oracle update batching, Oracle recommends you to keep the batch sizes in the general range of 50 to 100. This is because though the drivers support larger batches, they in turn result in a large memory footprint with no corresponding increase in performance. Very large batches usually result in a decline in performance compared to smaller batches.

Types of Statements Supported

As implemented by Oracle, update batching is intended for use with prepared statements, when you are repeating the same statement with different bind variables. Be aware of the following:

- Oracle update batching supports *only* prepared statement objects. For a callable statement, both the connection default batch value and the statement batch value are overridden with a value of 1. In an Oracle generic statement, there is no

statement batch value, and the connection default batch value is overridden with a value of 1.

- To adhere to the JDBC 2.0 standard, Oracle implementation of standard update batching supports callable statements, without `OUT` parameters, and generic statements, as well as prepared statements. You can migrate standard update batching into an Oracle JDBC application without difficulty.
- You can batch only `UPDATE`, `INSERT`, or `DELETE` operations. Processing a batch that includes an operation that attempts to return a result set will cause an exception.

Note: The Oracle implementation of standard update batching does not implement true batching for generic statements and callable statements. Although Oracle JDBC supports the use of standard batching syntax for `Statement` and `CallableStatement` objects, you will see performance improvement for only `PreparedStatement` objects.

Oracle Update Batching

The Oracle update batching feature associates a batch value with each prepared statement object. With Oracle update batching, instead of the JDBC driver running a prepared statement operation each time the `executeUpdate` method is called, the driver adds the statement to a batch of accumulated processing requests. The driver will pass all the operations to the database for processing once the batch value is reached. For example, if the batch value is 10, then each batch of 10 operations will be sent to the database and processed in one trip.

A method in the `OracleConnection` class enables you to set a default batch value for the Oracle connection as a whole, and this batch value applies to any Oracle prepared statement in the connection. For any particular Oracle prepared statement, a method in the `OraclePreparedStatement` class enables you to set a statement batch value that overrides the connection batch value. You can also override both batch values by choosing to manually process the pending batch.

Note:

- Do not mix standard update batching with Oracle update batching in the same application. The JDBC driver will throw an exception when you mix these.
- Disable auto-commit mode if you use update batching model. In case an error occurs while you are processing a batch, this provides you the option of committing or rolling back the operations that ran successfully prior to the error.

Oracle Update Batching Characteristics and Limitations

Note the following limitations and implementation details regarding Oracle update batching:

- By default, there is no statement batch value and the connection batch value is 1.
- Batch values between 5 and 30 tend to be the most effective. Setting a very high value might even have a negative effect. It is worth trying different values to verify the effectiveness for your particular application.

- Regardless of the batch value in effect, if any of the bind variables of an Oracle prepared statement is of a data type which requires data to be streamed, for instance LOB, CLOB, then Oracle JDBC driver sets the batch value to 1 and sends any queued requests to the database for processing.
- Oracle JDBC driver implicitly runs the `sendBatch` method of an Oracle prepared statement in any of the following circumstances:
  - The connection receives a `COMMIT` request as a result of calling the `commit` method.
  - The statement receives a `close` request.
  - The connection receives a `close` request.

  Note: A connection `COMMIT` request, statement `close`, or connection `close` has an effect on a pending batch only if you use Oracle update batching. However, if you use standard update batching, then it has no effect on a pending batch.

- If the connection receives a `ROLLBACK` request before `sendBatch` has been called, then the pending batched operations are not removed. You must explicitly call `clearBatch` to do this.

Setting the Connection Batch Value

You can specify a default batch value for any Oracle prepared statement in your Oracle connection. To do this, use the `setDefaultExecuteBatch` method of the `OracleConnection` object. For example, the following code sets the default batch value to 20 for all prepared statement objects associated with the `conn` connection object:

```
((OracleConnection)conn).setDefaultExecuteBatch(20);
```

The connection batch value will apply to statement objects created after this batch value was set.

Note that instead of calling the `setDefaultExecuteBatch` method, you can set the `defaultBatchValue` Java property if you use a Java `Properties` object in establishing the connection.

Setting the Statement Batch Value

Use the following steps to set the statement batch value for a particular Oracle prepared statement. This will override any connection batch value set using the `setDefaultExecuteBatch` method of the `OracleConnection` instance for the connection in which the statement is processed.

1. Write your prepared statement, and specify input values for the first row, as follows:

   ```
   PreparedStatement ps = conn.prepareStatement("INSERT INTO dept VALUES
   (?,?,?)");
   ps.setInt(1,12);
   ps.setString(2,"Oracle");
   ps.setString(3,"USA");
   ```

2. Cast your prepared statement to `OraclePreparedStatement`, and apply the `setExecuteBatch` method. In this example, the batch size of the statement is set to 2.

```
((OraclePreparedStatement)ps).setExecuteBatch(2);
```

If you wish, insert the `getExecuteBatch` method at any point in the program to check the default batch value for the statement, as follows:

```
System.out.println(" Statement Execute Batch Value " +
                   ((OraclePreparedStatement)ps).getExecuteBatch());
```

3. If you send an execute-update call to the database at this point, then no data will be sent to the database, and the call will return 0.

```
// No data is sent to the database by this call to executeUpdate
System.out.println("Number of rows inserted so far: " + ps.executeUpdate ());
```

4. If you enter a set of input values for a second row and an execute-update, then the number of batch calls to `executeUpdate` will be equal to the batch value of 2. The data will be sent to the database, and both rows will be inserted in a single round-trip. Since this executeUpdate call is at the batch value boundary, and all the operations in the batch are executed on the server, it will return the total number of affected rows by the whole batch.

```
ps.setInt(1, 11);
ps.setString(2, "Applications");
ps.setString(3, "Indonesia");

int rows = ps.executeUpdate();
System.out.println("Number of rows inserted now: " + rows);

ps.close();
```

Note: The batch value counter will reset internally if the data type for a column changes in between the batch. The total value of affected rows by the whole batch of previously batched items before the data type change, and the newly batched items will be returned at the new batch value boundary. For instance:

If the batch value is 8 after binding 4 integers to a `varchar2` column, a string is bound to the column. This will trigger an internal bind value counter reset, and if the application continues to bind strings, then the batch will be executed on the 12th `executeUpdate` call, and will return the affected row count of all 12 operations.

Checking the Batch Value

To check the overall connection batch value of an Oracle connection instance, use the `OracleConnection` class `getDefaultExecuteBatch` method:

```
Integer batch_val = ((OracleConnection)conn).getDefaultExecuteBatch();
```

To check the particular statement batch value of an Oracle prepared statement, use the `OraclePreparedStatement` class `getExecuteBatch` method:

```
Integer batch_val = ((OraclePreparedStatement)ps).getExecuteBatch();
```

Note: If no statement batch value has been set, then `getExecuteBatch` will return the connection batch value.

Overriding the Batch Value

If you want to process accumulated operations before the batch value in effect is reached, then use the `sendBatch` method of the `OraclePreparedStatement` object.

For this example, presume you set the connection batch value to 20. This sets the default batch value for all prepared statement objects associated with the connection to 20. You can accomplish this by casting your connection to `OracleConnection` and applying the `setDefaultExecuteBatch` method for the connection, as follows:

```
((OracleConnection)conn).setDefaultExecuteBatch (20);
```

Override the batch value as follows:

1. Write your prepared statement, specify input values for the first row, and then process the statement, as follows:

   ```
   PreparedStatement ps =
      conn.prepareStatement("insert into dept values (?, ?, ?)");

   ps.setInt(1, 32);
   ps.setString(2, "Oracle");
   ps.setString(3, "USA");

   System.out.println (ps.executeUpdate ());
   ```

   The batch is not processed at this point. The `ps.executeUpdate` method returns `0`.

2. If you enter a set of input values for a second operation and call `executeUpdate` again, then the data will still not be sent to the database, because the batch value in effect for the statement is the connection batch value, which is `20`.

   ```
   ps.setInt(1, 33);
   ps.setString(2, "Applications");
   ps.setString(3, "Indonesia");

   // this batch is still not executed at this point
   int rows = ps.executeUpdate();

   System.out.println("Number of rows updated before calling sendBatch: "  +
   rows);
   ```

   Note that the value of `rows` in the `println` statement is `0`.

3. If you apply the `sendBatch` method at this point, then the two previously batched operations will be sent to the database in a single round-trip. The `sendBatch` method also returns the total number of updated rows. This property of `sendBatch` is used by `println` to print the number of updated rows.

   ```
   // Execution of both previously batched executes will happen
   // at this point. The number of rows updated will be
   // returned by sendBatch.
   rows = ((OraclePreparedStatement)ps).sendBatch();

   System.out.println("Number of rows updated by calling sendBatch: " + rows);
   ps.close();
   ```

Committing the Changes in Oracle Batching

After you process the batch, you must still commit the changes, presuming auto-commit is disabled as recommended.

Calling the `commit` method on the connection object in Oracle batching not only commits operations in batches that have been processed, but also issues an implicit `sendBatch` call to process all pending batches. So, the `commit` method effectively commits changes for all operations that have been added to a batch.

Update Counts in Oracle Batching

In a nonbatching situation, the `executeUpdate` method of an `OraclePreparedStatement` object returns the number of database rows affected by the operation.

In an Oracle batching situation, this method returns the number of rows affected at the time the method is invoked, as follows:

- If an `executeUpdate` call results in the operation being added to the batch, then the method returns a value of 0, because nothing was written to the database yet.
- If an `executeUpdate` call results in the batch value being reached and the batch being processed, then the method returns the total number of rows affected by all operations in the batch.

Similarly, the `sendBatch` method of an `OraclePreparedStatement` object returns the total number of rows affected by all operations in the batch.

Example 21–1 illustrates the use of Oracle update batching.

Example 21–1 Oracle Update Batching

The following example illustrates how you use the Oracle update batching feature. It assumes you have imported the `oracle.driver.*` interfaces.

```
...
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci);
ods.setUser("HR");
ods.setPassword("hr");

Connection conn = ods.getConnection();
conn.setAutoCommit(false);

PreparedStatement ps =
  conn.prepareStatement("insert into dept values (?, ?, ?)");

//Change batch size for this statement to 3
((OraclePreparedStatement)ps).setExecuteBatch (3);

ps.setInt(1, 23);
ps.setString(2, "Sales");
ps.setString(3, "USA");
ps.executeUpdate(); //JDBC queues this for later execution

ps.setInt(1, 24);
ps.setString(2, "Blue Sky");
ps.setString(3, "Montana");
ps.executeUpdate(); //JDBC queues this for later execution

ps.setInt(1, 25);
ps.setString(2, "Applications");
ps.setString(3, "India");
ps.executeUpdate(); //The queue size equals the batch value of 3
                    //JDBC sends the requests to the database
```

```
ps.setInt(1, 26);
ps.setString(2, "HR");
ps.setString(3, "Mongolia");
ps.executeUpdate(); //JDBC queues this for later execution

((OraclePreparedStatement)ps).sendBatch(); // JDBC sends the queued request
conn.commit();

ps.close();
...
```

Note: Updates deferred through batching can affect the results of other queries. In the following example, if the first query is deferred due to batching, then the second will return unexpected results:

```
UPDATE emp SET name = "Sue" WHERE name = "Bob";
SELECT name FROM emp WHERE name = "Sue";
```

Error Reporting in Oracle Update Batching

If any one of the batched operations fails to complete successfully or attempts to return a result set during an `executeBatch` call, then the processing stops and a `java.sql.BatchUpdateException` is generated.

If the exception is raised, you can call the `getUpdateCounts` method on the `BatchUpdateException` object to retrieve the update count. This method returns an `int` array of update counts, just as the `executeBatch` method does.

Prior to Oracle Database 11g Release 1, the integer array returned contains *n* `Statement.EXECUTE_FAILED` entries, where *n* is the size of the batch. However, this does not indicate where in the batch the error occurred. The only option you have is to rollback the transaction.

Starting from Oracle Database 11*g* Release 1, the integer array returned contains *n* `Statement.SUCCESS_NO_INFO` entries, where *n* is the number of elements in the batch that have been successfully executed.

Note: The execution of the batch always stops with the first element of the batch that generates an error.

Standard Update Batching

JDBC standard update batching, unlike the Oracle update batching model, depends on explicitly adding statements to the batch using an `addBatch` method and explicitly processing the batch using an `executeBatch` method. In the Oracle model, you call `executeUpdate` as in a nonbatching situation, but whether an operation is added to the batch or the whole batch is processed is typically determined implicitly, depending on whether or not a predetermined batch value is reached.

Note:

- Do not mix standard update batching with Oracle update batching in the same application. Oracle JDBC driver will throw exceptions when these are mixed.
- Disable auto-commit mode if you use either update batching model. In case an error occurs while you are processing a batch, this provides you the option of committing or rolling back the operations that ran successfully prior to the error.

Limitations in the Oracle Implementation of Standard Batching

This section discusses the limitations and implementation details regarding the Oracle implementation of standard update batching.

In Oracle JDBC applications, update batching is intended for use with prepared statements that are being processed repeatedly with different sets of bind values.

The Oracle implementation of standard update batching does not implement true batching for generic statements and callable statements. Even though Oracle JDBC supports the use of standard batching for `Statement` and `CallableStatement` objects, you are unlikely to see performance improvement.

Adding Operations to the Batch

When any statement object is first created, its statement batch is empty. Use the standard `addBatch` method to add an operation to the statement batch. This method is specified in the standard `java.sql.Statement`, `PreparedStatement`, and `CallableStatement` interfaces, which are implemented by the `oracle.jdbc.OracleStatement`, `OraclePreparedStatement`, and `OracleCallableStatement` interfaces, respectively.

For a `Statement` object, the `addBatch` method takes a Java `String` with a SQL operation as input. For example:

```
...
Statement stmt = conn.createStatement();

stmt.addBatch("INSERT INTO emp VALUES(1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO dept VALUES(260, 'Sales')");
stmt.addBatch("INSERT INTO emp_dept VALUES(1000, 260)");
...
```

At this point, three operations are in the batch.

Note: Remember, however, that in the Oracle implementation of standard update batching, you will probably see no performance improvement in batching generic statements.

For prepared statements, update batching is used to batch multiple runs of the same statement with different sets of bind parameters. For a `PreparedStatement` or `OraclePreparedStatement` object, the `addBatch` method takes no input. It simply adds the operation to the batch using the bind parameters last set by the appropriate `set`*`XXX`* methods. This is also true for `CallableStatement` or `OracleCallableStatement` objects, but remember that in the Oracle implementation of standard update batching, you will probably see no performance improvement in batching callable statements.

For example:

```
...
PreparedStatement pstmt =
         conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();
...
```

At this point, two operations are in the batch.

Because a batch is associated with a single prepared statement object, you can batch only repeated runs of a single prepared statement, as in this example.

Processing the Batch

To process the current batch of operations, use the `executeBatch` method of the statement object. This method is specified in the standard `Statement` interface, which is extended by the standard `PreparedStatement` and `CallableStatement` interfaces.

> **Note:** If you add too many operations to a batch by calling the `addBatch` method several times and create a very large batch (for example, with more than or equal to 100,000 rows), then while calling the `executeBatch` method on the whole batch, you may face severe performance problems in terms of memory. To avoid this issue, the JDBC driver transparently breaks up the large batches into smaller internal batches and makes a roundtrip to the server for each internal batch. This makes your application slightly slower because of each round-trip overhead, but optimizes memory significantly. However, if each bound row is very large in size (for example, more than about 1MB each or so), then this process can impact the overall performance negatively because in such a case, the performance gained in terms of memory will be less than the performance lost in terms of time.

Following is an example that repeats the prepared statement `addBatch` calls shown previously and then processes the batch:

```
...
PreparedStatement pstmt =
         conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();

int[] updateCounts = pstmt.executeBatch();
...
```

Row Count per Iteration for Array DMLs

Starting from Oracle Database 12*c* Release 1 (12.1), the `executeBatch` method has been improved so that it returns an int array of size that is the same as the number of records in the batch and each item in the return array is the number of database table rows affected by the corresponding record of the batch. For example, if the batch size is 5, then the `executeBatch` method returns an array of size 5. In case of an error in between execution of the batch, the `executeBatch` method cannot return a value, instead it throws a `BatchUpdateException`. In this case, the exception itself carries an int array of size `n` as its data, where `n` is the number of successful record executions. For example, if the batch is of size 5 and the error occurs at the 4th record, then the `BatchUpdateException` has an array of size 3 (3 records executed successfully) and each item in the array represents how many rows were affected by each of them.

Committing the Changes in the Oracle Implementation of Standard Batching

After you process the batch, you must still commit the changes, presuming auto-commit is disabled as recommended.

Calling `commit`, commits nonbatched operations and batched operations for statement batches that have been processed, but for the Oracle implementation of standard batching, has no effect on pending statement batches that have *not* been processed.

Clearing the Batch

To clear the current batch of operations instead of processing it, use the `clearBatch` method of the statement object. This method is specified in the standard `Statement` interface, which is extended by the standard `PreparedStatement` and `CallableStatement` interfaces.

Keep the following things in mind:

- When a batch is processed, operations are performed in the order in which they were batched.
- After calling `addBatch`, you must call either `executeBatch` or `clearBatch` before a call to `executeUpdate`, otherwise there will be a SQL exception.
- A `clearBatch` or `executeBatch` call resets the statement batch to empty.
- The statement batch is not reset to empty if the connection receives a `ROLLBACK` request. You must explicitly call `clearBatch` to reset it.

Note:

- Oracle recommends not to use the `clearBatch` method with Oracle update batching. With Oracle update batching, binds can be implicitly flushed to the server at any point of time due to various conditions. This makes the behavior of the `clearBatch` method vary with each case and unpredictable at run time because the method cannot clear any bound data that is already flushed to the server. However, it is OK to use the `clearBatch` method after a rollback to clear any traces.

  Also, the `clearBatch` method displays this unpredictable behavior with only Oracle update batching and works correctly with standard update batching.

- If you are using Oracle update batching in Oracle Database 12*c* Release 1 (12.1), then you do not have to clear your batches explicitly in the code after a rollback. However, if you are using Oracle update batching in an earlier release, then you have to invoke the `clearBatch` method to clear your batches explicitly after a rollback.

- Invoking `clearBatch` method after a rollback works for all releases.
- An `executeBatch` call closes the current result set of the statement object, if one exists.
- Nothing is returned by the `clearBatch` method.

Following is an example that repeats the prepared statement `addBatch` calls shown previously but then clears the batch under certain circumstances:

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();

if (...condition...)
{
   int[] updateCounts = pstmt.executeBatch();
   ...
}
else
{
   pstmt.clearBatch();
   ...
}
```

Update Counts in the Oracle Implementation of Standard Batching

If a statement batch is processed successfully, then the integer array, or update counts array, returned by the statement `executeBatch` call will always have one element for each operation in the batch. In the Oracle implementation of standard update batching, the values of the array elements are as follows:

- For a prepared statement batch, the array contains the actual update counts indicating the number of rows affected by each operation.
- For a generic statement batch, the array contains the actual update counts indicating the number of rows affected by each operation. The actual update counts can be provided only in the case of generic statements in the Oracle implementation of standard batching.
- For a callable statement batch, the array contains the actual update counts indicating the number of rows affected by each operation.

In your code, upon successful processing of a batch, you should be prepared to handle either `-2`, `1`, or true update counts in the array elements. For a successful batch processing, the array contains either all `-2`, `1`, or all positive integers.

Example 21–2 illustrates the use of standard update batching.

Example 21–2 Standard Update Batching

This example combines the sample fragments in the previous sections, accomplishing the following steps:

1. Disabling auto-commit mode, which you should always do when using either update batching model
2. Creating a prepared statement object
3. Adding operations to the batch associated with the prepared statement object
4. Processing the batch
5. Committing the operations from the batch

```
conn.setAutoCommit(false);

PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();

int[] updateCounts = pstmt.executeBatch();

conn.commit();

pstmt.close();
...
```

You can process the update counts array to determine if the batch processed successfully.

Error Handling in the Oracle Implementation of Standard Batching

If any one of the batched operations fails to complete successfully or attempts to return a result set during an `executeBatch` call, then the processing stops and a `java.sql.BatchUpdateException` is generated.

After a batch exception, the update counts array can be retrieved using the `getUpdateCounts` method of the `BatchUpdateException` object. This returns an `int` array of update counts, just as the `executeBatch` method does. In the Oracle implementation of standard update batching, contents of the update counts array are as follows, after a batch is processed:

- For a prepared statement batch, in case of an error in between execution of the batch, the `executeBatch` method cannot return a value, instead it throws a `BatchUpdateException`. In this case, the exception itself carries an int array of size `n` as its data, where `n` is the number of successful record executions. For example, if the batch is of size 5 and the error occurs at the 4th record, then the `BatchUpdateException` has an array of size 3 (3 records executed successfully) and each item in the array represents how many rows were affected by each of them.
- For a generic statement batch or callable statement batch, the update counts array is only a partial array containing the actual update counts up to the point of the error. The actual update counts can be provided because Oracle JDBC cannot use true batching for generic and callable statements in the Oracle implementation of standard update batching.

  For example, if there were 20 operations in the batch, the first 13 succeeded, and the 14th generated an exception, then the update counts array will have 13 elements, containing actual update counts of the successful operations.

  You can either commit or roll back the successful operations in this situation, as you prefer.

In your code, upon failed processing of a batch, you should be prepared to handle either `-3` or true update counts in the array elements when an exception occurs. For a failed batch processing, you will have either a full array of `-3` or a partial array of positive integers.

Intermixing Batched Statements and Nonbatched Statements

You cannot call `executeUpdate` for regular, nonbatched processing of an operation if the statement object has a pending batch of operations.

However, you can intermix batched operations and nonbatched operations in a single statement object if you process nonbatched operations either prior to adding any operations to the statement batch or after processing the batch. Essentially, you can call `executeUpdate` for a statement object only when its update batch is empty. If the batch is non-empty, then an exception will be generated.

For example, it is valid to have a sequence, such as the following:

```
...
PreparedStatement pstmt =
         conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");

int scount = pstmt.executeUpdate();   // OK; no operations in pstmt batch

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
```

```
pstmt.addBatch();                    // Now start a batch

pstmt.setInt(1, 4000);
pstmt.setString(2, "Stan Leland");
pstmt.addBatch();

int[] bcounts = pstmt.executeBatch();

pstmt.setInt(1, 5000);
pstmt.setString(2, "Amy Feiner");

int scount = pstmt.executeUpdate();   // OK; pstmt batch was executed
...
```

Intermixing nonbatched operations on one statement object and batched operations on another statement object within your code is permissible. Different statement objects are independent of each other with regard to update batching operations. A `COMMIT` request will affect all nonbatched operations and all successful operations in processed batches, but will not affect any pending batches.

Premature Batch Flush

Premature batch flush happens due to a change in cached metadata. Cached metadata can be changed due to various reasons, such as the following:

- The initial bind was null and the following bind is not null.
- A scalar type is initially bound as string and then bound as scalar type or the reverse.

The premature batch flush count is summed to the return value of the next `executeUpdate` or `sendBatch` method.

The old functionality lost all these batch flush values which can be obtained now. To switch back to the old functionality, you can set the `AccumulateBatchResult` property to `false`, as follows:

```
java.util.Properties info = new java.util.Properties();
info.setProperty("user", "HR");
info.setProperty("passwd", "hr");
// other properties
...

// property: batch flush type
info.setProperty("AccumulateBatchResult", "false");

OracleDataSource ods = new OracleDataSource();
ods.setConnectionProperties(info);
ods.setURL("jdbc:oracle:oci:@");
Connection conn = ods.getConnection();
```

> **Note:** The `AccumulateBatchResult` property is set to `true` by default.

Example 21–3 illustrates premature batch flushing.

Example 21–3 Premature Batch Flushing

```
((OraclePreparedStatement)pstmt).setExecuteBatch (2);

pstmt.setNull(1, OracleTypes.NUMBER);
pstmt.setString(2, "test11");
int count = pstmt.executeUpdate(); // returns 0

/*
* Premature batch flush happens here.
*/
pstmt.setInt(1, 22);
pstmt.setString(2, "test22");
int count = pstmt.executeUpdate(); // returns 0

pstmt.setInt(1, 33);
pstmt.setString(2, "test33");
/*
*  returns 3 with the new batching scheme where as,
*  returns 2 with the old batching scheme.
*/
int count = pstmt.executeUpdate();
```

Additional Oracle Performance Extensions

In addition to update batching, Oracle JDBC drivers support the following extensions that improve performance by reducing round-trips to the database:

- Prefetching rows

  This reduces round-trips to the database by fetching multiple rows of data each time data is fetched. The extra data is stored in client-side buffers for later access by the client. The number of rows to prefetch can be set as desired.

- Specifying column types

  This avoids an inefficiency in the standard JDBC protocol for performing and returning the results of queries.

- Suppressing database metadata `TABLE_REMARKS` columns

  This avoids an expensive outer join operation.

Oracle provides several extensions to connection properties objects to support these performance extensions. These extensions enable you to set the `remarksReporting` flag and default values for row prefetching and update batching.

This section covers the following topics:

- Prefetching LOB Data
- Oracle Row-Prefetching Limitations
- Defining Column Types
- Reporting DatabaseMetaData TABLE_REMARKS

Prefetching LOB Data

For the JDBC drivers prior to Oracle Database 11*g* Release 2 JDBC drivers, if you want to retrieve `LOB` data in one round trip, then you have to fetch the data as `VARCHAR2` type, that is, you have to use `OracleTypes.VARCHAR` or `OracleTypes.LONGVARCHAR` with

the JDBC defineColumnType method. The limitation of this approach is that when LOB data is fetched as CHAR type, the locator cannot be fetched along with the data. So, if the application wants to get the LOB data at a later point of time, or if the application wants to perform other LOB operations, then one more round trip is required to get the LOB locator, as LOB locator is not available to the application.

Note: Array operations on LOB locators are not supported in the JDBC APIs.

Starting from Oracle Database 11*g* Release 2 JDBC drivers, the number of round trips is reduced by prefetching frequently used metadata, such as the LOB length and the chunk size as well as the beginning of the LOB data along with the locator during regular fetch operations. For small LOBs, the data may be totally prefetched in one single round trip, that is, the select parse, execution, and fetch occurs in one round trip, and performance is improved greatly. For large LOBs that are larger than 5 times the prefetch size, the performance improvement is not very significant as only the round trip for retrieving the chunk size is not needed.

defaultLobPrefetchSize Connection Property

Starting from Oracle Database 11*g* Release 2, there is a new connection property oracle.jdbc.defaultLobPrefetchSize that can be used to set the default LOB prefetch size for the connection. This connection property is defined as the following constant: OracleConnection.CONNECTION_PROPERTY_DEFAULT_LOB_PREFETCH_SIZE. The value of this property is used as the default LOB prefetch size for the current connection. The default value of this connection property is 4000. If you want to change the default value at the statement level, then use the setLobPrefetchSize method defined in oracle.jdbc.OracleStatement interface. You can change the default value to:

- -1 to disable LOB prefetch for the current connection
- 0 to enable LOB prefetch for metadata only
- Any value greater than 0 to specify the number of bytes for BLOBs and the number of characters for CLOBs to be prefetched along with the locator during fetch operations

Use getLobPrefetchSize method defined in oracle.jdbc.OracleStatement interface to retrieve the LOB prefetch size.

You can also set the value of LOB prefetch size at the column level by using the defineColumnType method. The column-level value overrides any value that is set at the connection or statement level.

See Also: The Javadoc for more information

Note: If LOB prefetch is not disabled at the connection level or statement level, it cannot be disabled at the column level.

Oracle Row-Prefetching Limitations

There is no maximum prefetch setting. The default value is 10. Larger or smaller values may be appropriate depending on the number of rows and columns expected from the query. You can set the default connection row-prefetch value using a Properties object.

When a statement object is created, it receives the default row-prefetch setting from the associated connection. Subsequent changes to the default connection row-prefetch setting will have no effect on the statement row-prefetch setting.

If a column of a result set is of data type `LONG`, `LONG RAW` or `LOB`s returned through the data interface, that is, the streaming types, then JDBC changes the statement row-prefetch setting to 1, even if you never actually read a value of either of these types.

Setting the prefetch size can affect the performance of an application. Increasing the prefetch size will reduce the number of round-trips required to get all the data, but will increase memory usage. This will depend on the number and size of the columns in the query and the number of rows expected to be returned. It will also depend on the memory and CPU loading of the JDBC client machine. The optimum for a standalone client application will be different from a heavily loaded application server. The speed and latency of the network connection should also be considered.

Note: Starting from Oracle Database 11*g* Release 1, the Thin driver can fetch the first `prefetch_size` number of rows from the server in the very first round-trip. This saves one round-trip in SELECT statements.

If you are migrating an application from earlier releases of Oracle JDBC drivers to 10*g* Release 1 (10.1) or later releases of Oracle JDBC drivers, then you should revisit the optimizations that you had done earlier, because the memory usage and performance characteristics may have changed substantially.

A common situation that you may encounter is, say, you have a query that selects a unique key. The query will return only zero or one row. Setting the prefetch size to 1 will decrease memory and CPU usage and cannot increase round-trips. However, you must be careful to avoid the error of requesting an extra fetch by writing `while(rs.next())` instead of `if(rs.next())`.

If you are using the JDBC Thin driver, then use the `useFetchSizeWithLongColumn` connection property, because it will perform `PARSE`, `EXECUTE`, and `FETCH` in a single round-trip.

Tuning of the prefetch size should be done along with tuning of memory management in your JVM under realistic loads of the actual application.

Note:

- Do not mix the JDBC 2.0 fetch size application programming interface (API) and the Oracle row-prefetching API in your application. You can use one or the other, but not both.
- Be aware that setting the Oracle fetch size value can affect not only queries, but also explicitly refetching rows in a result set through the result set `refreshRow` method, which is relevant for scroll-sensitive/read-only, scroll-sensitive/updatable, and scroll-insensitive/updatable result sets, and the window size of a scroll-sensitive result set, affecting how often automatic refetches are performed. However, the Oracle fetch size value will be overridden by any setting of the fetch size.

See Also: "Supported Connection Properties" on page 8-6

Defining Column Types

> **Note:** Starting from Oracle Database 12*c* Release 1 (12.1), the `defineColumnType` method is deprecated. For more information, refer to "Deprecated Features" on page 3-xxxi.

The implementation of `defineColumnType` changed significantly since Oracle Database 10*g*. Previously, `defineColumnType` was used both as a performance optimization and to force data type conversion. In previous releases, all of the drivers benefited from calls to `defineColumnType`. Starting from Oracle Database 10*g*, the JDBC Thin driver no longer needs the information provided. The JDBC Thin driver achieves maximum performance without calls to `defineColumnType`. The JDBC Oracle Call Interface (OCI) and server-side internal drivers still get better performance when the application uses `defineColumnType`.

If your code is used with both the JDBC Thin and OCI drivers, you can disable the `defineColumnType` method when using the Thin driver by setting the connection property `disableDefineColumnType` to `true`. Doing this makes `defineColumnType` have no effect. Do not set this connection property to `true` when using the JDBC OCI or server-side internal drivers.

You can also use `defineColumnType` to control how much memory the client-side allocates or to limit the size of variable-length data.

Follow these general steps to define column types for a query:

1. If necessary, cast your statement object to `OracleStatement`, `OraclePreparedStatement`, or `OracleCallableStatement`, as applicable.
2. If necessary, use the `clearDefines` method of your `Statement` object to clear any previous column definitions for this `Statement` object.
3. On each column, call the `defineColumnType` method of your `Statement` object, passing it these parameters:
   - Column index (integer)
   - Type code (integer)

     Use the `static` constants of the `java.sql.Types` class or `oracle.jdbc.OracleTypes` class, such as `Types.INTEGER`, `Types.FLOAT`, `Types.VARCHAR`, `OracleTypes.VARCHAR`, and `OracleTypes.ROWID`. Type codes for standard types are identical in these two classes.
   - Type name (string)

     For structured objects, object references, and arrays, you must also specify the type name. For example, `Employee`, `EmployeeRef`, or `EmployeeArray`.
   - Maximum field size (integer)

     Optionally specify a maximum data length for this column.

     You cannot specify a maximum field size parameter if you are defining the column type for a structured object, object reference, or array. If you try to include this parameter, it will be ignored.
   - Form of use (short)

     Optionally specify a form of use for the column. This can be `OraclePreparedStatement.FORM_CHAR` to use the database character set or

`OraclePreparedStatement.FORM_NCHAR` to use the national character set. If this parameter is omitted, the default is `FORM_CHAR`.

For example, assuming `stmt` is an Oracle statement, use:

```
stmt.defineColumnType(column_index, typeCode);
```

If the column is `VARCHAR` or equivalent and you know the length limit:

```
stmt.defineColumnType(column_index, typeCode, max_size);
```

For an `NVARCHAR` column where the original maximum length is desired and conversion to the database character set is requested:

```
stmt.defineColumnType(column_index, typeCode, 0,
   OraclePreparedStatement.FORM_CHAR );
```

For structured object, object reference, and array columns:

```
stmt.defineColumnType(column_index, typeCode, typeName);
```

Set a maximum field size if you do not want to receive the full default length of the data. Calling the `setMaxFieldSize` method of the standard JDBC `Statement` class sets a restriction on the amount of data returned. Specifically, the size of the data returned will be the minimum of the following:

- The maximum field size set in `defineColumnType`
- The maximum field size set in `setMaxFieldSize`
- The natural maximum size of the data type

After you complete these steps, use the `executeQuery` method of the statement to perform the query.

Note: It is no longer necessary to specify a data type for each column of the expected result set.

Example 21–4 illustrates the use of this feature. It assumes you have imported the `oracle.jdbc.*` interfaces.

Example 21–4 Defining Column Types

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@localhost:5221:orcl");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();

Statement stmt = conn.createStatement();
// Allocate only 2 chars for this column (truncation will happen)
((OracleStatement)stmt).defineColumnType(1, Types.VARCHAR, 2);
ResultSet rset = stmt.executeQuery("select ename from emp");
while(rset.next() )
  System.out.println(rset.getString(1));
stmt.close();
```

As this example shows, you must cast the `Statement` object, `stmt`, to `OracleStatement` in the invocation of the `defineColumnType` method. The `createStatement` method of the connection returns an object of type `java.sql.Statement`, which does not have the

defineColumnType and clearDefines methods. These methods are provided only in the OracleStatement implementation.

The define-extensions use JDBC types to specify the desired types. The allowed define types for columns depend on the internal Oracle type of the column.

All columns can be defined to their natural JDBC types. In most cases, they can be defined to the Types.CHAR or Types.VARCHAR type code.

Table 21–1 lists the valid column definition arguments you can use in the defineColumnType method.

Table 21–1 Valid Column Type Specifications

If the column has Oracle SQL type:	You can use defineColumnType to define it as:
NUMBER, VARNUM	BIGINT, TINYINT, SMALLINT, INTEGER, FLOAT, REAL, DOUBLE, NUMERIC, DECIMAL, CHAR, VARCHAR
CHAR, VARCHAR2	CHAR, VARCHAR
LONG	CHAR, VARCHAR, LONGVARCHAR
LONGRAW	LONGVARBINARY, VARBINARY, BINARY
RAW	VARBINARY, BINARY
DATE	DATE, TIME, TIMESTAMP, CHAR, VARCHAR
ROWID	ROWID
BLOB	VARBINARY, BINARY
CLOB	LONG, CHAR, VARCHAR

It is always valid to use defineColumnType with the original data type of the column.

Reporting DatabaseMetaData TABLE_REMARKS

The getColumns, getProcedureColumns, getProcedures, and getTables methods of the database metadata classes are slow if they must report TABLE_REMARKS columns, because this necessitates an expensive outer join. For this reason, the JDBC driver does *not* report TABLE_REMARKS columns by default.

You can enable TABLE_REMARKS reporting by passing a true argument to the setRemarksReporting method of an OracleConnection object.

Equivalently, instead of calling setRemarksReporting, you can set the remarksReporting Java property if you use a Java Properties object in establishing the connection.

If you are using a standard java.sql.Connection object, you must cast it to OracleConnection to use setRemarksReporting.

Example 21–5 illustrates how to enable TABLE_REMARKS reporting.

Example 21–5 TABLE_REMARKS Reporting

Assuming conn is the name of your standard Connection object, the following statement enables TABLE_REMARKS reporting:

```
((oracle.jdbc.OracleConnection)conn ).setRemarksReporting(true);
```

Considerations for getColumns

By default, the `getColumns` method does not retrieve information about the columns if a synonym is specified. To enable the retrieval of information if a synonym is specified, you must call the `setIncludeSynonyms` method on the connection as follows:

```
((oracle.jdbc.OracleConnection)conn ).setIncludeSynonyms(true)
```

This will cause all subsequent `getColumns` method calls on the connection to include synonyms. This is similar to `setRemarksReporting`. Alternatively, you can set the `includeSynonyms` connection property. This is similar to the `remarksReporting` connection property.

However, bear in mind that if `includeSynonyms` is `true`, then the name of the object returned in the `table_name` column will be the synonym name, if a synonym exists. This is true even if you pass the table name to `getColumns`.

Considerations for getProcedures and getProcedureColumns Methods

According to JDBC versions 1.1 and 1.2, the methods `getProcedures` and `getProcedureColumns` treat the `catalog`, `schemaPattern`, `columnNamePattern`, and `procedureNamePattern` parameters in the same way. In the Oracle definition of these methods, the parameters are treated differently:

- `catalog`

  Oracle does not have multiple catalogs, but it does have packages. Consequently, the `catalog` parameter is treated as the package name. This applies both on input, which is the `catalog` parameter, and the output, which is the `catalog` column in the returned `ResultSet`. On input, the construct `" "`, which is an empty string, retrieves procedures and arguments without a package, that is, standalone objects. A `null` value means to drop from the selection criteria, that is, return information about both standalone and packaged objects. That is, it has the same effect as passing in the percent sign (`%`). Otherwise, the `catalog` parameter should be a package name pattern, with SQL wild cards, if desired.

- `schemaPattern`

  All objects within Oracle database must have a schema, so it does not make sense to return information for those objects without one. Thus, the construct `" "`, which is an empty string, is interpreted on input to mean the objects in the current schema, that is, the one to which you are currently connected. To be consistent with the behavior of the `catalog` parameter, `null` is interpreted to drop the schema from the selection criteria. That is, it has the same effect as passing in `%`. It can also be used as a pattern with SQL wild cards.

- `procedureNamePattern` and `columnNamePattern`

  The empty string (" ") does not make sense for either parameter, because all procedures and arguments must have names. Thus, the construct `" "` will raise an exception. To be consistent with the behavior of other parameters, `null` has the same effect as passing in percent sign (`%`).

22

OCI Connection Pooling

The Java Database Connectivity (JDBC) Oracle Call Interface (OCI) driver connection pooling functionality is part of the JDBC client. This functionality is provided by the `OracleOCIConnectionPool` class.

A JDBC application can have multiple pools at the same time. Multiple pools can correspond to multiple application servers or pools to different data sources. The connection pooling provided by the JDBC OCI driver enables applications to have multiple logical connections, all using a small set of physical connections. Each call on a logical connection gets routed on to the physical connection that is available at the time of call.

This chapter contains the following sections:

Note: Use OCI connection pooling if you need session multiplexing. Otherwise, Oracle recommends using Universal Connection Pool.

OCI Driver Connection Pooling: Background

The Oracle JDBC OCI driver provides several transaction monitor capabilities, such as the fine-grained management of Oracle sessions and connections. It is possible for a high-end application server or transaction monitor to multiplex several sessions over fewer physical connections on a call-level basis, thereby achieving a high degree of scalability by pooling of connections and back-end Oracle server processes.

The connection pooling provided by the `OracleOCIConnectionPool` interface simplifies the session/connection separation interface hiding the management of the physical connection pool. The Oracle sessions are the `OracleOCIConnection` objects obtained from `OracleOCIConnectionPool`. The connection pool itself is usually configured with a much smaller shared pool of physical connections, translating to a back-end server pool containing an identical number of dedicated server processes.

Note that many more Oracle sessions can be multiplexed over this pool of fewer shared connections and back-end Oracle processes.

OCI Driver Connection Pooling and Shared Servers Compared

In some ways, what OCI driver connection pooling offers on the middle tier is similar to what shared server processes offer on the back end. OCI driver connection pooling makes a dedicated server instance behaves as a shared instance by managing the session multiplexing logic on the middle tier. Therefore, the pooling of dedicated server processes and incoming connections into the dedicated server processes is controlled by the OCI connection pool on the middle tier.

The main difference between OCI connection pooling and shared servers is that in the case of shared servers, the connection from the client is typically to a dispatcher in the database instance. The dispatcher is responsible for directing the client request to an appropriate shared server. On the other hand, the physical connection from the OCI connection pool is established directly from the middle tier to the Oracle dedicated server process in the back-end server pool.

Note that OCI connection pool is mainly beneficial only if the middle tier is multithreaded. Each thread could maintain a session to the database. The actual connections to the database are maintained by `OracleOCIConnectionPool`, and these connections, including the pool of dedicated database server processes, are shared among all the threads in the middle tier.

Defining an OCI Connection Pool

An OCI connection pool is created at the beginning of the application. Creating connections from a pool is quite similar to creating connections using the `OracleDataSource` class.

The `oracle.jdbc.pool.OracleOCIConnectionPool` class, which extends the `OracleDataSource` class, is used to create OCI connection pools. From an `OracleOCIConnectionPool` instance, you can obtain logical connection objects. These connection objects are of the `OracleOCIConnection` class type. This class implements the `OracleConnection` interface. The `Statement` objects you create from the `OracleOCIConnection` instance have the same fields and methods as `OracleStatement` objects you create from `OracleConnection` instances.

The following code shows header information for the `OracleOCIConnectionPool` class:

```
/*
   * @param us  ConnectionPool user-id.
   * @param p   ConnectionPool password
   * @param name  logical name of the pool. This needs to be one in the
   *                   tnsnames.ora configuration file.
     @param config (optional)  Properties of the pool, if the default does not
                suffice. Default connection configuration is min =1, max=1,
                incr=0
                  Please refer setPoolConfig for property names.

                 Since this is optional, pass null if the default configuration
                 suffices.

   * @return
   *
   * Notes: Choose a userid and password that can act as proxy for the users
   *        in the getProxyConnection() method.
```

```
            If config is null, then the following default values will take
            effect
            CONNPOOL_MIN_LIMIT = 1
            CONNPOOL_MAX_LIMIT = 1
            CONNPOOL_INCREMENT = 0

*/

public synchronized OracleOCIConnectionPool
  (String       user,    String   password,  String name, Properties config)
  throws SQLException

/*
 * This will use the user-id, password and connection pool name values set
   LATER using the methods setUser, setPassword, setConnectionPoolName.

 * @return
 *
 * Notes:

     No OracleOCIConnection objects can be created on
     this class unless the methods setUser, setPassword, setPoolConfig
     are invoked.
     When invoking the setUser, setPassword later, choose a userid and
     password that can act as proxy for the users
 *   in the getProxyConnection() method.
 */
  public synchronized OracleOCIConnectionPool ()
    throws SQLException
```

Importing the oracle.jdbc.pool and oracle.jdbc.oci Packages

Before you create an OCI connection pool, import the following to have Oracle OCI connection pooling functionality:

```
import oracle.jdbc.pool.*;
import oracle.jdbc.oci.*;
```

Creating an OCI Connection Pool

The following code show how you create an instance of the `OracleOCIConnectionPool` class called `cpool`:

```
OracleOCIConnectionPool cpool = new OracleOCIConnectionPool
    ("HR", "hr", "jdbc:oracle:oci:@(description=(address=(host=
    localhost)(protocol=tcp)(port=5221))(connect_data=(INSTANCE_NAME=orcl)))",
    poolConfig);
```

`poolConfig` is a set of properties that specify the connection pool. If `poolConfig` is null, then the default values are used. For example, consider the following:

- `poolConfig.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "4");`
- `poolConfig.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "10");`
- `poolConfig.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "2");`

As an alternative to the constructor call, you can create an instance of the `OracleOCIConnectionPool` class using individual methods to specify the user, password, and connection string.

```
OracleOCIConnectionPool cpool = new OracleOCIConnectionPool ( );
cpool.setUser("HR");
cpool.setPassword("hr");
cpool.setURL("jdbc:oracle:oci:@(description=(address=(host=
    localhost)(protocol=tcp)(port=5221))(connect_data=(INSTANCE_NAME=orcl)))");
cpool.setPoolConfig(poolConfig);  // In case you want to specify a different
                                  // configuration other than the default
                                  // values.
```

Setting the OCI Connection Pool Parameters

The connection pool configuration is determined by the following `OracleOCIConnectionPool` class attributes:

- `CONNPOOL_MIN_LIMIT`

  Specifies the minimum number of physical connections that can be maintained by the pool.

- `CONNPOOL_MAX_LIMIT`

  Specifies the maximum number of physical connections that can be maintained by the pool.

- `CONNPOOL_INCREMENT`

  Specifies the incremental number of physical connections to be opened when all the existing ones are busy and a call needs one more connection; the increment is done only when the total number of open physical connections is less than the maximum number that can be opened in that pool.

- `CONNPOOL_TIMEOUT`

  Specifies how much time must pass before an idle physical connection is disconnected; this does not affect a logical connection.

- `CONNPOOL_NOWAIT`

  Specifies, if enabled, that an error is returned if a call needs a physical connection while the maximum number of connections in the pool are busy. If disabled, a call waits until a connection is available. Once this attribute is set to `true`, it cannot be reset to `false`.

You can configure all of these attributes dynamically. Therefore, an application has the flexibility of reading the current load, that is number of open connections and number of busy connections, and adjusting these attributes appropriately, using the `setPoolConfig` method.

> **Note:** The default values for the `CONNPOOL_MIN_LIMIT`, `CONNPOOL_MAX_LIMIT`, and `CONNPOOL_INCREMENT` parameters are `1`, `1`, and `0`, respectively.

The `setPoolConfig` method is used to configure OCI connection pool properties. The following is a typical example of how the `OracleOCIConnectionPool` class attributes can be set:

```
...
java.util.Properties p  = new java.util.Properties( );
p.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "1");
p.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "5");
p.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "2");
```

```
p.put (OracleOCIConnectionPool.CONNPOOL_TIMEOUT, "10");
p.put (OracleOCIConnectionPool.CONNPOOL_NOWAIT, "true");
cpool.setPoolConfig(p);
...
```

Observe the following rules when setting these attributes:

- `CONNPOOL_MIN_LIMIT`, `CONNPOOL_MAX_LIMIT`, and `CONNPOOL_INCREMENT` are mandatory.
- `CONNPOOL_MIN_LIMIT` must be a value greater than zero.
- `CONNPOOL_MAX_LIMIT` must be a value greater than or equal to `CONNPOOL_MIN_LIMIT` plus `CONNPOOL_INCREMENT`.
- `CONNPOOL_INCREMENT` must be a value greater than or equal to zero.
- `CONNPOOL_TIMEOUT` must be a value greater than zero.
- `CONNPOOL_NOWAIT` must be `true` or `false`.

See Also: *Oracle Call Interface Programmer's Guide*

Checking the OCI Connection Pool Status

To check the status of the connection pool, use the following methods from the `OracleOCIConnectionPool` class:

- `int getMinLimit()`

  Retrieves the minimum number of physical connections that can be maintained by the pool.

- `int getMaxLimit()`

  Retrieves the maximum number of physical connections that can be maintained by the pool.

- `int getConnectionIncrement()`

  Retrieves the incremental number of physical connections to be opened when all the existing ones are busy and a call needs a connection.

- `int getTimeout()`

  Retrieves the specified time (in seconds) that a physical connection in a pool can remain idle before it is disconnected; the age of a connection is based on the Least Recently Used (LRU) algorithm.

- `String getNoWait()`

  Retrieves if the `NOWAIT` property is enabled. It returns a string of "`true`" or "`false`".

- `int getPoolSize()`

  Retrieves the number of physical connections that are open. This should be used only as an estimate and for statistical analysis.

- `int getActiveSize()`

  Retrieves the number of physical connections that are open and busy. This should be used only as an estimate and for statistical analysis.

- `boolean isPoolCreated()`

  Retrieves if the pool has been created. The pool is actually created when `OracleOCIConnection(user, password, url, poolConfig)` is called or when

setUser, setPassword, and setURL has been done after calling OracleOCIConnection().

Connecting to an OCI Connection Pool

The OracleOCIConnectionPool class, through a getConnection method call, creates an instance of the OracleOCIConnection class. This instance represents a connection.

Because the OracleOCIConnection class extends OracleConnection class, it has the functionality of this class too. Close the OracleOCIConnection objects once the user session is over, otherwise, they are closed when the pool instance is closed.

There are two ways of calling getConnection:

- OracleConnection getConnection()

  If you do not supply the user name and password, then the default user name and password used for the creation of the connection pool are used while creating the connection objects.

- OracleConnection getConnection(String user, String password)

  If you this method, you will get a logical connection identified with the specified user name and password, which can be different from that used for pool creation.

The following code shows the signatures of the overloaded getConnection method:

```
public synchronized OracleConnection getConnection( )
                   throws SQLException

/*
 * For getting a connection to the database.
 *
 * @param us  Connection user-id
 * @param p   Connection password
 * @return     connection object
 */
public synchronized OracleConnection getConnection(String us, String p)
throws SQLException
```

As an enhancement to OracleConnection, the following new method is added into OracleOCIConnection as a way to change the password for the user:

```
void passwordChange (String user, String oldPassword, String newPassword)
```

Sample Code for OCI Connection Pooling

The following code illustrates the use of OCI connection pooling in a sample application:

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import oracle.jdbc.OracleDriver;
import oracle.jdbc.pool.OracleOCIConnectionPool;

public class conPoolAppl extends Thread
{
```

```
  public static final String query = "SELECT object_name FROM all_objects WHERE
rownum < 300";
  static public void main(String args[]) throws SQLException
  {
    int _maxCount = 10;
    Connection []conn = new Connection[_maxCount];
    try
    {
      String s = null;   //System.getProperty ("JDBC_URL");
      String url = "jdbc:oracle:oci8:@localhost";
      OracleOCIConnectionPool cpool = new OracleOCIConnectionPool("HR", "hr", url,
null);

      // Print out the default configuration for the OracleOCIConnectionPool
      System.out.println ("-- The default configuration for the
OracleOCIConnectionPool --");
      displayPoolConfig(cpool);

      //Set up the initial pool configuration
      Properties p1  = new Properties();
      p1.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, Integer.toString(1));
      p1.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, Integer.toString(_
maxCount));
      p1.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, Integer.toString(1));

      // Enable the initial configuration
      cpool.setPoolConfig(p1);

      Thread []t = new Thread[_maxCount];
      for (int i = 0; i < _maxCount; ++i)
      {
        conn[i] = cpool.getConnection("HR", "hr");
        if ( conn[i] == null )
        {
          System.out.println("Unable to create connection.");
          return;
        }
        t[i] = new conPoolAppl (i, conn[i]);
        t[i].start ();
        //displayPoolConfig(cpool);
      }

      ((conPoolAppl)t[0]).startAllThreads ();
      try
      {
        Thread.sleep (200);
      }
      catch (Exception ea) {}

      displayPoolConfig(cpool);
      for (int i = 0; i < _maxCount; ++i)
        t[i].join ();
    }
    catch(Exception ex)
    {
      System.out.println("Error: " + ex);
      ex.printStackTrace ();
      return;
    }
    finally
```

```
  {
    for (int i = 0; i < _maxCount; ++i)
      if (conn[i] != null)
      conn[i].close ();
  }
} //end of main

private Connection m_conn;
private static boolean m_startThread = false;
private int m_threadId;

public conPoolAppl (int i, Connection conn)
{
  m_threadId = i;
  m_conn = conn;
}

public void startAllThreads ()
{
  m_startThread = true;
}

public void run ()
{
  while (!m_startThread) Thread.yield ();
  try
  {
    doQuery (m_conn);
  }
  catch (SQLException ea)
  {
    System.out.println ("*** Thread id: " + m_threadId);
    ea.printStackTrace ();
  }
} // end of run

private static void doQuery (Connection conn) throws SQLException
{
  PreparedStatement pstmt = null;
  ResultSet rs = null;
  try
  {
    pstmt = conn.prepareStatement (query);
    rs = pstmt.executeQuery ();
    while (rs.next ())
    {
      //System.out.println ("Object name: " +rs.getString (1));
    }
  }
  catch (Exception ea)
  {
    System.out.println ("Error during execution: " +ea);
    ea.printStackTrace ();
  }
  finally
  {
    if (rs != null)
      rs.close ();
    if (pstmt != null)
      pstmt.close ();
```

```
      if (conn != null)
        conn.close ();
    }
  }  // end of doQuery (Connection)

  // Display the current status of the OracleOCIConnectionPool
  private static void displayPoolConfig (OracleOCIConnectionPool cpool) throws
SQLException
  {
    System.out.println (" Min poolsize Limit: " + cpool.getMinLimit());
    System.out.println (" Max poolsize Limit: " + cpool.getMaxLimit());
  /*
    System.out.println (" Connection Increment: " +
cpool.getConnectionIncrement());
    System.out.println (" NoWait: " + cpool.getNoWait());
    System.out.println (" Timeout: " + cpool.getTimeout());
  */
    System.out.println (" PoolSize: " + cpool.getPoolSize());
    System.out.println (" ActiveSize: " + cpool.getActiveSize());
  }

}  // end of class conPoolAppl
```

Statement Handling and Caching

Statement caching is supported with `OracleOCIConnectionPool`. The caching improves performance by not having to open, parse, and close cursors. When `OracleOCIConnection.prepareStatement` ("*a_SQL_query*") is processed, the statement cache is searched for a statement that matches the SQL query. If a match is found, then you can reuse the `Statement` object instead of incurring the cost of creating another `Statement` object. The cache size can be dynamically increased or decreased. The default cache size is zero.

Note: The `OracleStatement` object created from `OracleOCIConnection` has the same behavior as one that is created from `OracleConnection`.

JNDI and the OCI Connection Pool

The Java Naming and Directory Interface (JNDI) feature makes the properties of a Java object persist, therefore these properties can be used to construct a new instance of the object, such as cloning the object. The benefit is that the old object can be freed, and at a later time a new object with exactly the same properties can be created. The `InitialContext.bind` method makes the properties persist, either on file or in a database, while the `InitialContext.lookup` method retrieves the properties from the persistent store and creates a new object with these properties.

`OracleOCIConnectionPool` objects can be bound and looked up using the JNDI feature. No new interface calls in `OracleOCIConnectionPool` are necessary.

23

Database Resident Connection Pooling

Database Resident Connection Pool (DRCP) is a connection pool in the server that is shared across many clients. You should use DRCP in connection pools where the number of active connections is fairly less than the number of open connections. As the number of instances of connection pools that can share the connections from DRCP pool increases, the benefits derived from using DRCP increases. DRCP increases Database server scalability and resolves the resource wastage issue that is associated with middle-tier connection pooling.

This chapter contains the following sections:

Overview of Database Resident Connection Pooling

In middle-tier connection pools, every connection cache maintains a minimum number of connections to the server. Each connection represents used up resources at the server. All these open connections are not utilized at any given time, which means that there are unused resources that unnecessarily take up server resources. In a multiple middle-tier scenario, these connections are not shared with any other middle tier and are retained in the cache even if some of these are idle. However, a large number of such middle-tier connection pools increase the number of inactive connections to the Database server significantly and waste a lot of Database resources because all the connections do not remain active simultaneously.

For example, in a middle-tier connection pool, if the minimum pool size is 200, then the connection pool has 200 connections to the server, and the Database server has 200 server processes associated with these connections. If there are 30 middle tiers with a connection pool of minimum size 200, then the server has 6000 (200 * 30) corresponding server processes running. Typically, on an average only 5% of the connections, and in turn, server processes are in use at any given time. So, out of the 6,000 server processes, only 300 server processes are active at any given time. This leads to over 5,700 unused server processes on the server. These unused processes are the wasted resources on the server.

The Database Resident Connection Pool implementation creates a pool on the server side, which is shared across multiple client pools. This significantly lowers memory consumption on the server because of reduced number of server processes on the server and increases the scalability of the Database server.

See Also:

- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide*
- *Oracle Database PL/SQL Packages and Types Reference*

Enabling Database Resident Connection Pooling

This section describes how to enable DRCP in the server side and the client side:

- Enabling DRCP on the Server Side
- Enabling DRCP on the Client Side

Enabling DRCP on the Server Side

You must be a database administrator (DBA) and must log on as `SYSDBA` to start and end a pool. This section contains the following subsections:

- Setting the Statement Cache Size
- Starting a Pool
- Ending a Pool

Note: The features of DRCP can be leveraged only with a connection pool on the client because JDBC does not have a default pool on its own. If you do not have a client connection pool and make any change to the Database with auto commit set to `false`, then the changes are not committed to the Database while closing the connection.

Starting a Pool

Run the `dbms_connection_pool.start_pool` method with the default settings to start the default pool, `SYS_DEFAULT_CONNECTION_POOL`. For example:

```
sqlplus /nolog
connect / as sysdba
execute dbms_connection_pool.start_pool();
```

See Also: *Oracle Database Administrator's Guide* and "About Statement Caching" for more information

Ending a Pool

Run the `dbms_connection_pool.stop_pool` method with the default settings to end the pool. For example:

```
sqlplus /nolog
connect / as sysdba
execute dbms_connection_pool.stop_pool();
```

Setting the Statement Cache Size

If you use DRCP, caching is also done at the server side. So, you must specify the statement cache size on the server side in the following way, where 50 is the preferred size:

```
execute DBMS_CONNECTION_POOL.CONFIGURE_POOL (session_cached_cursors=>50);
```

Enabling DRCP on the Client Side

Perform the following steps to enable DRCP on the client side:

> **Note:** In Example 23–1, we are using Universal Connection Pool as the client-side connection pool. For any other connection pools, you can enable DRCP by performing the following two steps and using `oracle.jdbc.pool.OracleConnectionPoolDataSource` as the connection factory.

- Pass a non-null and non-empty String value to the connection property `oracle.jdbc.DRCPConnectionClass`
- Append (`SERVER=POOLED`) to the `CONNECT_DATA` in the long connection string

You can also specify `(SERVER=POOLED)` in short URL from as follows:

```
jdbc:oracle:thin:@//<host>:<port>/<service_name>[:POOLED]
```

For example:

```
jdbc:oracle:thin:@//localhost:5221/orcl:POOLED
```

Example 23–1 shows how to enable DRCP on client side:

Example 23–1 Enabling DRCP on Client Side Using Universal Connection Pool

```
String url = "jdbc:oracle:thin:@//localhost:5521/orcl:POOLED";
PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
// Set DataSource Property
pds.setUser("HR");
pds.setPassword("hr");
System.out.println ("Connecting to " + url);
pds.setURL(url);
pds.setConnectionPoolName("HR-Pool1");
pds.setMinPoolSize(2);
pds.setMaxPoolSize(3);
pds.setInitialPoolSize(2);
Properties prop = new Properties();
prop.put("oracle.jdbc.DRCPConnectionClass", "HR-Pool1");
pds.setConnectionProperties(prop);
```

> **See Also:** Oracle Universal Connection Pool for JDBC Developer's Guide for more information about Universal Connection Pool

> **Note:** In UCP, if you do not provide a connection class, then the connection pool name is used as the connection class name by default.

Sharing Pooled Servers Across Multiple Connection Pools

To share pooled server processes on the server across multiple Connection pools, set the same DRCP Connection class name for all the pooled server processes on the

server. You can set the DRCP Connection class name using the connection property `oracle.jdbc.DRCPConnectionClass` as discussed in "Enabling DRCP on the Client Side" on page 23-3.

DRCP Tagging

DRCP enables you to request the Server connection pool to associate a Server process with a particular tag name.

> **See Also:** *Oracle Call Interface Programmer's Guide* for more information about session pooling and connection tagging

APIs for Using DRCP

In DRCP, you can apply a tag to a given connection and retrieve that tagged connection later. Connection tagging enhances session pooling because you can retrieve specific sessions easily.

If you want to take advantage of DRCP with higher granular control for your Custom connection pool implementations, then you must use the following APIs declared in the `oracle.jdbc.OracleConnection` interfaces:

- `attachServerConnection`
- `detachServerConnection`
- `isDRCPEnabled`
- `needToPurgeStatementCache`

> **See Also:** *Oracle Database JDBC Java API Reference*

24
Oracle Advanced Queuing

Oracle Advanced Queuing (AQ) provides database-integrated message queuing functionality. It is built on top of Oracle Streams and optimizes the functions of Oracle Database so that messages can be stored persistently, propagated between queues on different computers and databases, and transmitted using Oracle Net Services, HTTP, and HTTPS. Because Oracle AQ is implemented in database tables, all operational benefits of high availability, scalability, and reliability are also applicable to queue data. This chapter provides information about the Java interface to Oracle AQ.

Note:

- Oracle Advanced Queuing (AQ) is a feature of the Oracle JDBC Thin driver and is not supported by JDBC OCI driver.
- In Oracle Database 12*c* Release 1 (12.1), support for `XMLType` queues has been added. Till Oracle Database 11*g* Release 1, supported queue types were `RAW`, `ADT`, and `ANYDATA` queue types.

See Also: *Oracle Database Advanced Queuing User's Guide*

This chapters covers the following topics:

Functionality and Framework of Oracle Advanced Queuing

The Oracle JDBC package `oracle.jdbc.aq` provides a fast Java interface to AQ. This package contains the following:

- Classes
  - `AQDequeueOptions`

    Specifies the options available for the dequeue operation

  - `AQEnqueueOptions`

    Specifies the options available for the enqueue operation

  - `AQFactory`

    Is a factory class for AQ

  - `AQNotificationEvent`

    Is created whenever a new message is enqueued in a queue for which you have registered your interest

- Interfaces
  - `AQAgent`

    Used to represent and identify a user of the queue or a producer or consumer of the message

  - `AQMessage`

    Represents a message that is enqueued or dequeued

  - `AQMessageProperties`

    Contains message properties such as Correlation, Sender, Delay and Expiration, Recipients, and Priority and Ordering

  - `AQNotificationListener`

    Is a listener interface for receiving AQ notification events

  - `AQNotificationRegistration`

    Represents your interest in being notified when a new message is enqueued in a particular queue

These classes and interfaces enable you to access an existing queue, create messages, and enqueue and dequeue messages.

Note: Oracle JDBC drivers do *not* provide any API to create a queue. Queues must be created through the `DBMS_AQADM` PL/SQL package.

See Also: For more information about the APIs, refer to *Oracle Database JDBC Java API Reference*.

Making Changes to the Database

The code snippets used in this chapter assume that user `HR` is connecting to the database. Therefore, in the database, you must grant the following privileges to `HR`:

```
GRANT EXECUTE ON DBMS_AQ to HR;
GRANT EXECUTE ON DBMS_AQADM to HR;
GRANT AQ_ADMINISTRATOR_ROLE TO HR;
GRANT ADMINISTER DATABASE TRIGGER TO HR;
```

Before you start enqueuing and dequeuing messages, you must have queues in the Database. For this, you must perform the following:

1. Create a queue table in the following way:

```
BEGIN
    DBMS_AQADM.CREATE_QUEUE_TABLE(
```

```
            QUEUE_TABLE =>'HR.RAW_SINGLE_QUEUE_TABLE',
            QUEUE_PAYLOAD_TYPE =>'RAW',
            COMPATIBLE => '10.0');
END;
```

2. Create a queue in the following way:

```
BEGIN
    DBMS_AQADM.CREATE_QUEUE(
            QUEUE_NAME =>'HR.RAW_SINGLE_QUEUE',
            QUEUE_TABLE =>'HR.RAW_SINGLE_QUEUE_TABLE',
END;
```

3. Start the queue in the following way:

```
BEGIN
    DBMS_AQADM.START_QUEUE(
 'HR.RAW_SINGLE_QUEUE',
END;
```

It is a good practice to stop the queue and remove the queue tables from the database. You can perform this in the following way:

1. Stop the queue in the following way:

```
BEGIN
    DBMS_AQADM.STOP_QUEUE(
 HR.RAW_SINGLE_QUEUE',
END;
```

2. Remove the queue tables from the database in the following way:

```
BEGIN
    DBMS_AQADM.DROP_QUEUE_TABLE(
            QUEUE_TABLE =>'HR.RAW_SINGLE_QUEUE_TABLE',
            FORCE => TRUE
END;
```

AQ Asynchronous Event Notification

A JDBC application can do the following:

- Register to the AQ namespace and receive notification when an enqueue occurs. This can be performed in the following way:

```
 public AQNotificationRegistration registerForAQEvents(
    OracleConnection conn,
    String queueName) throws SQLException
  {
    Properties globalOptions = new Properties();
    String[] queueNameArr = new String[1];
    queueNameArr[0] = queueName;
    Properties[] opt = new Properties[1];
    opt[0] = new Properties();
    opt[0].setProperty(OracleConnection.NTF_AQ_PAYLOAD,"true");
    AQNotificationRegistration[] regArr =
conn.registerAQNotification(queueNameArr,opt,globalOptions);
    AQNotificationRegistration reg = regArr[0];
    return reg;
  }
```

- Register subscriptions to database events and receive notifications when the events are triggered

  Registered clients are notified asynchronously when events are triggered or on an explicit AQ enqueue (or a new message is enqueued in a queue for which you have registered your interest). Clients do not need to be connected to a database.

  The following code snippet shows how to subscribe to database events and receive notifications when the events are triggered:

```
class DemoAQRawQueueListener implements AQNotificationListener
{
  OracleConnection conn;
  String queueName;
  String typeName;
  int eventsCount = 0;

  public DemoAQRawQueueListener(String _queueName, String _typeName)
   throws SQLException
  {
   queueName = _queueName;
   typeName = _typeName;
   conn = (OracleConnection)DriverManager.getConnection
     (DemoAQRawQueue.URL, DemoAQRawQueue.USERNAME, DemoAQRawQueue.PASSWORD);
  }

  public void onAQNotification(AQNotificationEvent e)
  {
    try
    {
     AQDequeueOptions deqopt = new AQDequeueOptions();
     deqopt.setRetrieveMessageId(true);
     if(e.getConsumerName() != null)
       deqopt.setConsumerName(e.getConsumerName());
     if((e.getMessageProperties()).getDeliveryMode()
        == AQMessageProperties.DeliveryMode.BUFFERED)
     {
       deqopt.setDeliveryMode(AQDequeueOptions.DEQUEUE_BUFFERED);
       deqopt.setVisibility(AQDequeueOptions.DEQUEUE_IMMEDIATE);
     }
     AQMessage msg = conn.dequeue(queueName,deqopt,typeName);
     byte[] msgId = msg.getMessageId();
     if(msgId != null)
     {
       String mesgIdStr = DemoAQRawQueue.byteBufferToHexString(msgId,20);
       System.out.println("ID of message dequeued = "+mesgIdStr);
     }
     System.out.println(msg.getMessageProperties().toString());
     byte[] payload = msg.getPayload();
     if(typeName.equals("RAW"))
     {
       String payloadStr = new String(payload,0,10);
       System.out.println("payload.length="+payload.length+",
value="+payloadStr);
     }
    }
    catch(SQLException sqlex)
    {
     System.out.println(sqlex.getMessage());
    }
    eventsCount++;
```

```
  }
  public int getEventsCount()
  {
    return eventsCount;
  }
  public void closeConnection() throws SQLException
  {
    conn.close();
  }
}
```

- Register to the listener in the following way:

```
AQNotificationRegistration reg = registerForAQEvents(conn,queueName+":BLUE");
DemoAQRawQueueListener demo_li = new
DemoAQRawQueueListener(queueName,queueType);
reg.addListener(demo_li);
```

Creating Messages

Before you enqueue a message, you must create the message. An instance of a class implementing the `AQMessage` interface represents an AQ message. An AQ message contains properties (metadata) and a payload (data). Perform the following to create an AQ message:

1. Create an instance of `AQMessageProperties` in the following way:

```
AQMessageProperties msgprop = AQFactory.createAQMessageProperties();
```

2. Set the property attributes in the following way:

```
msgprop.setCorrelation("mycorrelation");
msgprop.setExceptionQueue("MY_EXCEPTION_QUEUE");
msgprop.setExpiration(0);
msgprop.setPriority(1);
```

3. Create the AQ message using the `AQMessageProperties` object in the following way:

```
AQMessage mesg = AQFactory.createAQMessage(msgprop);
```

4. Set the payload in the following way:

```
byte[] rawPayload = "Example_Payload".getBytes();
mesg.setPayload(new oracle.sql.RAW(rawPayload));
```

AQ Message Properties

The properties of the AQ message are represented by an instance of the `AQMessageProperties` interface. You can set or get the following message properties:

- Dequeue Attempts Count: Specifies the number of attempts that have been made to dequeue the message. This property cannot be set.
- Correlation: Is an identifier supplied by the producer of the message at the time of enqueuing the message.
- Delay: Is the number of seconds for which the message is in the `WAITING` state. After the specified delay, the message is in the `READY` state and available for

dequeuing. Dequeuing a message by using the message ID (msgid) overrides the delay specification.

> **Note:** Delay is not supported with buffered messaging.

- Delivery Mode: Specifies whether the message is a buffered message or a persistent message. This property cannot be set.
- Enqueue Time: Specifies the time at which the message was enqueued. This value is determined by the system and cannot be set by the user.
- Exception Queue: Specifies the name of the queue into which the message is moved if it cannot be processed successfully. Messages are moved in two cases:
  - The number of unsuccessful dequeue attempts has exceeded `max_retries`.
  - The message has expired.
- Expiration: Is the number of seconds during which the message is available for dequeuing, starting from when the message reaches the `READY` state. If the message is not dequeued before it expires, then it is moved to the exception queue in the `EXPIRED` state.
- Message State: Specifies the state of the message at the time of dequeuing the message. This property cannot be set.
- Previous Queue Message ID: Is the ID of the message in the last queue that generated the current message. When a message is propagated from one queue to another, this attribute identifies the ID of the queue from which it was last propagated. This property cannot be set.
- Priority: Specifies the priority of the message. It can be any integer including negative integers; the smaller the value, the higher the priority.
- Recipient list: Is a list of `AQAgent` objects that represent the recipients. The default recipients are the queue subscribers. This parameter is valid only for multiple-consumer queues.
- Sender: Is an identifier specified by the producer at the time of enqueuing the message. It is an instance of `AQAgent`.
- Transaction group: Specifies the transaction group of the message for transaction-grouped queues. It is set after a successful call to the `dequeueArray` method.

AQ Message Payload

Depending on the type of the queue, the payload of the AQ message can be specified using the `setPayload` method of the `AQMessage` interface. The following code snippet illustrates how to set the payload:

```
...
byte[] rawPayload = "Example_Payload".getBytes();
mesg.setPayload(new oracle.sql.RAW(rawPayload));
...
```

You can retrieve the payload of an AQ message using the `getPayload` method or the appropriate `get`*`XXX`*`Payload` method in the following way:

```
byte[] payload = mesg.getPayload();
```

These methods are defined in the `AQMessage` interface.

Example: Creating a Message and Setting a Payload

This section provides an example that illustrates how to create a message and set a payload.

Example 24–1 Creating a Message and Setting a Payload

This example shows how to Create an instance of `AQMessageProperties`, set the property attributes, create the AQ message, and set the payload.

```
 AQMessageProperties msgprop = AQFactory.createAQMessageProperties();
    msgprop.setCorrelation("mycorrelation");
    msgprop.setExceptionQueue("MY_EXCEPTION_QUEUE");
    AQAgent ag = AQFactory.createAQAgent();
    ag.setName("MY_SENDER_AGENT_NAME");
    ag.setAddress("MY_SENDER_AGENT_ADDRESS");
    msgprop.setSender(ag);
    // handle multi consumer case:
    if(recipients != null)
      msgprop.setRecipientList(recipients);
    System.out.println(msgprop.toString());
    AQMessage mesg = AQFactory.createAQMessage(msgprop);
byte[] rawPayload = "Example_Payload".getBytes();
mesg.setPayload(new oracle.sql.RAW(rawPayload));
```

Enqueuing Messages

After you create a message and set the message properties and payload, you can enqueue the message using the `enqueue` method of the `OracleConnection` interface. Before you enqueue the message, you can specify some enqueue options. The `AQEnqueueOptions` class enables you to specify the following enqueue options:

- Delivery mode: Specifies the delivery mode. Delivery mode can be set to either persistent (`ENQUEUE_PERSISTENT`) or buffered (`ENQUEUE_BUFFERED`).
- Retrieve Message ID: Specifies whether or not the message ID has to be retrieved from the server when the message has been enqueued. By default, the message ID is not retrieved.
- Transformation: Specifies a transformation that will be applied before enqueuing the message. The return type of the transformation function must match the type of the queue.

  Note: Transformations must be created in PL/SQL using `DBMS_TRANSFORM.CREATE_TRANSFORMATION(...)`.

- Visibility: Specifies the transactional behavior of the enqueue request. The default value for this option is `ENQUEUE_ON_COMMIT`. It indicates that the enqueue operation is part of the current transaction. `ENQUEUE_IMMEDIATE` indicates that the enqueue operation is an autonomous transaction, which commits at the end of the operation. For buffered messaging, you must use `ENQUEUE_IMMEDIATE`.

The following code snippet illustrates how to set the enqueue options and enqueue the message:

```
...
AQEnqueueOptions opt = new AQEnqueueOptions();
opt.setRetrieveMessageId(true);
```

```
conn.enqueue(queueName, opt, mesg);
...
```

Dequeuing Messages

Enqueued messages can be dequeued using the `dequeue` method of the `OracleConnection` interface. Before you dequeue a message you must set the dequeue options. The `AQDequeueOptions` class enables you to specify the following dequeue options:

- Condition: Specifies a conditional expression based on the message properties, the message data properties, and PL/SQL functions. A dequeue condition is specified as a `Boolean` expression using syntax similar to the `WHERE` clause of a SQL query.
- Consumer name: If specified, only the messages matching the consumer name are accessed.

  Note: If the queue is a single-consumer queue, do *not* set this option.

- Correlation: Specifies a correlation criterion (or search criterion) for the dequeue operation.
- Delivery Filter: Specifies the type of message to be dequeued. You dequeue buffered messages only (`DEQUEUE_BUFFERED`) or persistent messages only (`DEQUEUE_PERSISTENT`), which is the default, or both (`DEQUEUE_PERSISTENT_OR_BUFFERED`).
- Dequeue Message ID: Specifies the message identifier of the message to be dequeued. This can be used to dequeue a unique message whose ID is known.
- Dequeue mode: Specifies the locking behavior associated with the dequeue operation. It can take one of the following values:
  - `DequeueMode.BROWSE`: Message is dequeued without acquiring any lock.
  - `DequeueMode.LOCKED`: Message is dequeued with a write lock that lasts for the duration of the transaction.
  - `DequeueMode.REMOVE`: (default) Message is dequeued and deleted. The message can be retained in the queue based on the retention properties.
  - `DequeueMode.REMOVE_NO_DATA`: Message is marked as updated or deleted.
- Maximum Buffer Length: Specifies the maximum number of bytes that will be allocated when dequeuing a message from a `RAW` queue. The default maximum is `DEFAULT_MAX_PAYLOAD_LENGTH` but it can be changed to any other nonzero value. If the buffer is not large enough to contain the entire message, then the exceeding bytes will be silently ignored.
- Navigation: Specifies the position of the message that will be retrieved. It can take one of the following values:
  - `NavigationOption.FIRST_MESSAGE`: The first available message matching the search criteria is dequeued.
  - `NavigationOption.NEXT_MESSAGE`: (default) The next available message matching the search criteria is dequeued. If the previous message belongs to a message group, then the next available message matching the search criteria in the message group is dequeued.

  - `NavigationOption.NEXT_TRANSACTION`: Messages in the current transaction group are skipped, and the first message of the next transaction group is dequeued. This setting can be used *only* if message grouping is enabled for the queue.

- Retrieve Message ID: Specifies whether or not the message identifier of the dequeued message needs to be retrieved. By default, it is not retrieved.
- Transformation: Specifies a transformation that will be applied after dequeuing the message. The source type of the transformation must match the type of the queue.

  Note: Transformations must be created in PL/SQL using `DBMS_TRANSFORM.CREATE_TRANSFORMATION(...)`.

- Visibility: Specifies whether or not the message is dequeued as part of the current transaction. It can take one of the following values:
  - `VisibilityOption.ON_COMMIT`: (default) The dequeue operation is part of the current transaction.
  - `VisibilityOption.IMMEDIATE`: The dequeue operation is an autonomous transaction that commits at the end of the operation.

  Note: The Visibility option is ignored in the `DequeueMode.BROWSE` dequeue mode. If the delivery filter is `DEQUEUE_BUFFERED` or `DEQUEUE_PERSISTENT_OR_BUFFERED`, then this option *must* be set to `VisibilityOption.IMMEDIATE`.

- Wait: Specifies the wait time for the dequeue operation, if none of the messages matches the search criteria. The default value is `DEQUEUE_WAIT_FOREVER` indicating that the operation waits forever. If set to `DEQUEUE_NO_WAIT`, then the operation does not wait. If a number is specified, then the dequeue operation waits for the specified number of seconds.

  Note: If you use `DEQUEUE_WAIT_FOREVER`, then the dequeue operation will not return until a message that matches the search criterion is available in the queue. However, you can interrupt the dequeue operation by calling the `cancel` method on the `OracleConnection` object.

The following code snippet illustrates how to set the dequeue options and dequeue the message:

```
...
AQDequeueOptions deqopt = new AQDequeueOptions();
deqopt.setRetrieveMessageId(true);
deqopt.setConsumerName(consumerName);
AQMessage msg = conn.dequeue(queueName,deqopt,queueType);
```

Examples: Enqueuing and Dequeuing

This section provides a few examples that illustrate how to enqueue and dequeue messages.

Example 24–2 illustrates how to enqueue a message, and Example 24–3 illustrates how to dequeue a message.

Example 24–2 Enqueuing a Single Message

This example illustrates how to obtain access to a queue, create a message, and enqueue it.

```
AQMessageProperties msgprop = AQFactory.createAQMessageProperties();
msgprop.setPriority(1);
msgprop.setExceptionQueue("EXCEPTION_QUEUE");
msgprop.setExpiration(0);
AQAgent agent = AQFactory.createAQAgent();
agent.setName("AGENTNAME");
agent.setAddress("AGENTADDRESS");
msgprop.setSender(agent);
AQMessage mesg = AQFactory.createAQMessage(msgprop);
mesg.setPayload(buffer); // where buffer is a byte array (for a RAW queue)
AQEnqueueOptions options = new AQEnqueueOptions();
conn.enqueue("HR.MY_QUEUE", options, mesg);
```

Example 24–3 Dequeuing a Single Message

This example illustrates how to obtain access to a queue, set the dequeue options, and dequeue the message.

```
AQDequeueOptions options = new AQDequeueOptions();
options.setDeliveryFilter(AQDequeueOptions.DeliveryFilter.BUFFERED);
AQMessage mesg = conn.dequeue("HR.MY_QUEUE", options, "RAW");
```

25
Continuous Query Notification

Generally, a middle-tier data cache duplicates some data from the back-end database server. Its goal is to avoid redundant queries to the database. However, this is efficient only when the data rarely changes in the database. The data cache has to be updated or invalidated when the data changes in the database. Starting from 11*g* Release 1, Oracle JDBC drivers provide support for the Continuous Query Notification feature of Oracle Database. Using this functionality, multitier systems can take advantage of the Continuous Query Notification feature to maintain a data cache as up-to-date as possible, by receiving invalidation events from the JDBC drivers.

> **Note:** Continuous Query Notification is a feature of the Oracle JDBC Thin driver and is not supported by JDBC OCI driver.

The JDBC drivers can register SQL queries with the database and receive notifications in response to the following:

- DML or DDL changes on the objects associated with the queries
- DML or DDL changes that affect the result set

The notifications are published when the DML or DDL transaction commits (changes made in a local transaction do not generate any event until they are committed).

To use Oracle JDBC driver support for Continuous Query Notification, perform the following:

1. Registration: You first need to create a registration.
2. Query association: After you have created a registration, you can associate SQL queries with it. These queries are part of the registration.
3. Notification: Notifications are created in response to changes in tables or result set. Oracle database communicates these notifications to the JDBC drivers through a dedicated network connection and JDBC drivers convert these notifications to Java events.

Also, you need to grant the `CHANGE NOTIFICATION` privilege to the user. For example, if you connect to the database using the `HR` user name, then you need to run the following command in the database:

```
grant change notification to HR;
```

This section describes the following topics:

Creating a Registration

Creating a registration is a one-time process and is done outside the currently used transaction. The API for creating a registration in the server is executed in its own transaction and is committed immediately. You need a JDBC connection to create a registration, however, the registration is not attached to the connection. You can close the connection after creating a registration, and the registration survives. In an Oracle RAC environment, a registration is a persistent entity that exists on all nodes. The registration exists in the Database. So, even if a node goes down, the registration continues to exist and is notified when the tables change.

There are two ways to create a registration:

- The JDBC-style of registration: Use the JDBC driver to create a registration on the server. The JDBC driver launches a new thread that listens to notifications from the server (through a dedicated channel) and converts these notification messages into Java events. The driver then notifies all the listeners registered with this registration.
- The PL/SQL-style of registration: If you want a PL/SQL stored procedure to handle the notifications, then create a PL/SQL-style registration. As in the JDBC-style of registration, the JDBC drivers enable you to attach statements (queries) to this registration. However the JDBC drivers do not get notifications from the server because the notifications are handled by the PL/SQL stored procedure.

> **Note:** This approach is useful only for nonmultithreaded languages, such as PHP.

There is no way to remove one particular object (table) from an existing registration. A workaround would be to either create a new registration without this object or ignore the events that are related to this object.

You can use the `registerDatabaseChangeNotification` method of the `oracle.jdbc.OracleConnection` interface to create a JDBC-style of registration. You can set certain registration options through the `options` parameter of this method. Table 25–1 lists some of the registration options that can be set. To set these options, use the `java.util.Properties` object. These options are defined in the `oracle.jdbc.OracleConnection` interface. The registration options have a direct impact on the notification events that the JDBC drivers will create. Example 25–1 illustrates how to use the Continuous Query Notification feature.

The `registerDatabaseChangeNotification` method creates a new database change registration in the database server with the given options. It returns a `DatabaseChangeRegistration` object, which can then be used to associate a statement with this registration. It also opens a listener socket that will be used by the database to send notifications.

> **Note:** If a listener socket (created by a different registration) exists, then this socket will be used by the new database change registration as well.

Table 25–1 Continuous Query Notification Registration Options

Option	Description
DCN_IGNORE_DELETEOP	If set to true, DELETE operations will not generate any database change event.
DCN_IGNORE_INSERTOP	If set to true, INSERT operations will not generate any database change event.
DCN_IGNORE_UPDATEOP	If set to true, UPDATE operations will not generate any database change event.
DCN_NOTIFY_CHANGELAG	Specifies the number of transactions by which the client is willing to lag behind. **Note:** If this option is set to any value other than 0, then ROWID level granularity of information will not be available in the events, even if the DCN_NOTIFY_ROWIDS option is set to true.
DCN_NOTIFY_ROWIDS	Database change events will include row-level details, such as operation type and ROWID.
DCN_QUERY_CHANGE_NOTIFICATION	Activates query change notification instead of object change notification. **Note:** This option is available only when running against an 11.0 database.
NTF_LOCAL_HOST	Specifies the IP address of the computer that will receive the notifications from the server.
NTF_LOCAL_TCP_PORT	Specifies the TCP port that the driver should use for the listener socket.
NTF_QOS_PURGE_ON_NTFN	Specifies if the registration should be expunged on the first notification event.
NTF_QOS_RELIABLE	Specifies whether or not to make the notifications persistent, which comes at a performance cost.
NTF_TIMEOUT	Specifies the time in seconds after which the registration will be automatically expunged by the database.

If there exists a registration, then you can also use the `getDatabaseChangeRegistration` method to map the existing registration with a new `DatabaseChangeRegistration` object. This method is particularly useful if you have created a registration using PL/SQL and want to associate a statement with it.

See: Refer to the Javadoc for more information about the APIs.

Associating a Query with a Registration

After you have created a registration or mapped to an existing registration, you can associate a query with it. Like creating a registration, associating a query with a registration is a one-time process and is done outside of the currently used registration. The query will be associated even if the local transaction is rolled back.

You can associate a query with registration using the `setDatabaseChangeRegistration` method defined in the `OracleStatement` class. This method takes a `DatabaseChangeRegistration` object as parameter. The following code snippet illustrates how to associate a query with a registration:

```
...
// conn is an OracleConnection object.
// prop is a Properties object containing the registration options.
DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotifictaion(prop);
...
Statement stmt = conn.createStatement();
```

```
// associating the query with the registration
((OracleStatement)stmt).setDatabaseChangeRegistration(dcr);
// any query that will be executed with the 'stmt' object will be associated with
// the registration 'dcr' until 'stmt' is closed or
// '((OracleStatement)stmt).setDatabaseChangeRegistration(null);' is executed.
...
```

Notifying Database Change Events

To receive Continuous Query Notifications, attach a listener to the registration. When a database change event occurs, the database server notifies the JDBC driver. The driver then constructs a new Java event, identifies the registration to be notified, and notifies the listeners attached to the registration. The event contains the object ID of the database object that has changed and the type of operation that caused the change. Depending on the registration options, the event may also contain row-level detail information. The listener code can then use the event to make decisions about the data cache.

> **Note:** The listener code must not slow down the JDBC notification mechanism. If the code is time-consuming, for example, if it refreshes the data cache by querying the database, then it needs to be executed within its own thread.

You can attach a listener to a registration using the `addListener` method. The following code snippet illustrates how to attach a listener to a registration:

```
...
// conn is an OracleConnection object.
// prop is a Properties object containing the registration options.
DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotifictaion(prop);
...
// Attach the listener to the registration.
// Note: DCNListener is a custom listener and not a predefined or standard
// lsiener
DCNListener list = new DCNListener();
dcr.addListener(list);
...
```

Deleting a Registration

You need to explicitly unregister a registration to delete it from the server and release the resources in the driver. You can unregister a registration using a connection different from one that was used for creating it. To unregister a registration, you can use the `unregisterDatabaseChangeNotification` method defined in `oracle.jdbc.OracleConnection`.

You must pass the `DatabaseChangeRegistration` object as a parameter to this method. This method deletes the registration from the server and the driver and closes the listener socket.

If the registration was created outside of JDBC, say using PL/SQL, then you must pass the registration ID instead of the `DatabaseChangeRegistration` object. The method will delete the registration from the server, however, it does not free any resources in the driver.

Example

Example 25–1 illustrates how to use the Continuous Query Notification feature. In this example, the HR user is connecting to the database. Therefore in the database you need to grant the following privilege to the user:

```
grant change notification to HR;
```

Example 25–1 Continuous Query Notification

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.jdbc.OracleStatement;
import oracle.jdbc.dcn.DatabaseChangeEvent;
import oracle.jdbc.dcn.DatabaseChangeListener;
import oracle.jdbc.dcn.DatabaseChangeRegistration;

public class DBChangeNotification
{
  static final String USERNAME= "HR";
  static final String PASSWORD= "hr";
  static String URL;

  public static void main(String[] argv)
  {
    if(argv.length < 1)
    {
      System.out.println("Error: You need to provide the URL in the first
argument.");
      System.out.println("  For example: > java -classpath .:ojdbc6.jar
DBChangeNotification \"jdbc:oracle:thin:
@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=yourhost.yourdomain.com)(PORT=5221))(CO
NNECT_DATA=
(SERVICE_NAME=orcl)))\"");

      System.exit(1);
    }
    URL = argv[0];
    DBChangeNotification demo = new DBChangeNotification();
    try
    {
      demo.run();
    }
    catch(SQLException mainSQLException )
    {
      mainSQLException.printStackTrace();
    }
  }

  void run() throws SQLException
  {
    OracleConnection conn = connect();

    // first step: create a registration on the server:
    Properties prop = new Properties();
```

```
    // if connected through the VPN, you need to provide the TCP address of the
client.
    // For example:
    // prop.setProperty(OracleConnection.NTF_LOCAL_HOST,"14.14.13.12");

    // Ask the server to send the ROWIDs as part of the DCN events (small
performance
    // cost):
    prop.setProperty(OracleConnection.DCN_NOTIFY_ROWIDS,"true");
//
//Set the DCN_QUERY_CHANGE_NOTIFICATION option for query registration with finer
granularity.
 prop.setProperty(OracleConnection.DCN_QUERY_CHANGE_NOTIFICATION,"true");

    // The following operation does a roundtrip to the database to create a new
    // registration for DCN. It sends the client address (ip address and port)
that
    // the server will use to connect to the client and send the notification
    // when necessary. Note that for now the registration is empty (we haven't
registered
    // any table). This also opens a new thread in the drivers. This thread will
be
    // dedicated to DCN (accept connection to the server and dispatch the events
to
    // the listeners).
    DatabaseChangeRegistration dcr =
conn.registerDatabaseChangeNotification(prop);

    try
    {
      // add the listenerr:
      DCNDemoListener list = new DCNDemoListener(this);
      dcr.addListener(list);

      // second step: add objects in the registration:
      Statement stmt = conn.createStatement();
      // associate the statement with the registration:
      ((OracleStatement)stmt).setDatabaseChangeRegistration(dcr);
      ResultSet rs = stmt.executeQuery("select * from dept where deptno='45'");
      while (rs.next())
      {}
      String[] tableNames = dcr.getTables();
      for(int i=0;i<tableNames.length;i++)
        System.out.println(tableNames[i]+" is part of the registration.");
      rs.close();
      stmt.close();
    }
    catch(SQLException ex)
    {
      // if an exception occurs, we need to close the registration in order
      // to interrupt the thread otherwise it will be hanging around.
      if(conn != null)
        conn.unregisterDatabaseChangeNotification(dcr);
      throw ex;
    }
    finally
    {
      try
      {
        // Note that we close the connection!
```

```
        conn.close();
      }
      catch(Exception innerex){ innerex.printStackTrace(); }
    }

    synchronized( this )
    {
      // The following code modifies the dept table and commits:
      try
      {
        OracleConnection conn2 = connect();
        conn2.setAutoCommit(false);
        Statement stmt2 = conn2.createStatement();
        stmt2.executeUpdate("insert into dept (deptno,dname) values ('45','cool
dept')",
Statement.RETURN_GENERATED_KEYS);
        ResultSet autoGeneratedKey = stmt2.getGeneratedKeys();
        if(autoGeneratedKey.next())
          System.out.println("inserted one row with
ROWID="+autoGeneratedKey.getString(1));
        stmt2.executeUpdate("insert into dept (deptno,dname) values ('50','fun
dept')",
Statement.RETURN_GENERATED_KEYS);
        autoGeneratedKey = stmt2.getGeneratedKeys();
        if(autoGeneratedKey.next())
          System.out.println("inserted one row with
ROWID="+autoGeneratedKey.getString(1));
        stmt2.close();
        conn2.commit();
        conn2.close();
      }
      catch(SQLException ex) { ex.printStackTrace(); }

      // wait until we get the event
      try{ this.wait();} catch( InterruptedException ie ) {}
    }

    // At the end: close the registration (comment out these 3 lines in order
    // to leave the registration open).
    OracleConnection conn3 = connect();
    conn3.unregisterDatabaseChangeNotification(dcr);
    conn3.close();
  }

  /**
   * Creates a connection the database.
   */
  OracleConnection connect() throws SQLException
  {
    OracleDriver dr = new OracleDriver();
    Properties prop = new Properties();
    prop.setProperty("user",DBChangeNotification.USERNAME);
    prop.setProperty("password",DBChangeNotification.PASSWORD);
    return (OracleConnection)dr.connect(DBChangeNotification.URL,prop);
  }
}
/**
 * DCN listener: it prints out the event details in stdout.
 */
class DCNDemoListener implements DatabaseChangeListener
```

```
{
  DBChangeNotification demo;
  DCNDemoListener(DBChangeNotification dem)
  {
    demo = dem;
  }
  public void onDatabaseChangeNotification(DatabaseChangeEvent e)
  {
    Thread t = Thread.currentThread();
    System.out.println("DCNDemoListener: got an event ("+this+" running on thread
"+t+")");
    System.out.println(e.toString());
    synchronized( demo ){ demo.notify();}
  }
}
```

This code will also work with Oracle Database 10*g* Release 2 (10.2). This code uses table registration. That is, when you register a `SELECT` query, what you register is the name of the tables involved and not the query itself. In other words, you might select one single row of a table and if another row is updated, you will be notified although the result of your query has not changed.

In this example, if you leave the registration open instead of closing it, then the Continuous Query Notification thread continues to run. Now if you run a DML query that changes the `HR.DEPARTMENTS` table and commit it, say from SQL*Plus, then the Java program prints the notification.

Part VI

High Availability

This section provides information about two new features of Oracle Database 12*c* Release 1 (12.1), namely Transaction Guard for Java and Application Continuity for Java, along with the Transparent Application Failover (TAF) feature.

Part VI contains the following chapters:

26

Transaction Guard for Java

Oracle Database 12*c* Release 1 (12.1) introduces Transaction Guard feature that provides a generic infrastructure for at-most-once execution during planned and unplanned outages and duplicate submissions. This chapter discusses Transaction Guard for Java in the following sections:

Overview of Transaction Guard for Java

For the current applications, determining the outcome of the last commit operation in a guaranteed and scalable manner, following a communication failure to the server, is an unsolved problem. In many cases, the end users are asked to follow certain steps to avoid resubmitting duplicate request. For example, some applications warn users not to click the Submit button twice because if it is not followed, then users may unintentionally purchase the same items twice and submit multiple payments for the same invoice.

To solve this problem, Transaction Guard for Java provides transaction idempotence, that is, every transaction has at-most-once execution that prevents applications from submitting duplicate transactions. Every transaction is tagged with a Logical Transaction Identifier (`LTXID`), which can be used by the application after the occurrence of a failure to verify whether the transaction had committed before the failure or not. For example, if the commit calls do not return, then, using the `LTXID`, the application can find out whether it succeeded or not.

See Also: *Oracle Database Development Guide*

The Application Continuity for Java feature uses Transaction Guard for Java internally, which enables transparent session recovery and replay of SQL statements (queries and DMLs) since the beginning of the in-flight transaction. Application Continuity enables recovery of work after the occurrence of a planned or unplanned outage and Transaction Guard for Java ensures transaction idempotence. When an outage occurs, the recovery restores the state exactly as it was before the failure occurred.

See Also: "Application Continuity for Java"

Transaction Guard for Java APIs

This section discusses the APIs associated with Transaction Guard for Java for the following activities:

- Retrieving the Logical Transaction Identifiers
- Retrieving the Updated Logical Transaction Identifiers

Retrieving the Logical Transaction Identifiers

Use the `getLogicalTransactionId` method of the `oracle.jdbc.OracleConnection` interface to retrieve the current Logical Transaction Identifiers that are sent by the server. This method call does not make a database round-trip.

Example

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// Getting the 1st LTXID after connecting
LogicalTransactionId firstLtxid = oconn.getLogicalTransactionId();
```

Retrieving the Updated Logical Transaction Identifiers

Use the `oracle.jdbc.LogicalTransactionIdEventListener` interface for receiving updates to Logical Transaction Identifiers. You must implement this interface in your application to process the Logical Transaction Identifier events.

Registering Event Listeners

Use the `addLogicalTransactionIdEventListener` method to register a listener to the Logical Transaction Identifier events.

Example

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// The subsequent LTXID updates can be obtained through the listener
oconn.addLogicalTransactionIdEventListener(this);
```

You can also use the `addLogicalTransactionIdEventListener(LogicalTransactionIdEventListener listener, java.util.concurrent.Executor executor)` method to register a listener with an executor.

Unregistering Event Listeners

Use the `removeLogicalTransactionIdEventListener` method to unregister a listener from the Logical Transaction Identifier events.

Example

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
```

```
// The subsequent LTXID updates can be obtained through the listener
oconn.removeLogicalTransactionIdEventListener(this);
```

Using Transaction Guard APIs: Complete Example

The following is a complete example using the Transaction Guard APIs.

```
import oracle.jdbc.pool.OracleDataSource;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.LogicalTransactionId;
import oracle.jdbc.LogicalTransactionIdEvent;
import oracle.jdbc.LogicalTransactionIdEventListener;

public class transactionGuardExample
{
      ...
      ...
      OracleDataSource ods = new OracleDataSource();
      ods.setURL(url);
      ods.setUser("user");
      ods.setPassword("password");

      OracleConnection oconn = (OracleConnection) ods.getConnection();

      // Getting the 1st LTXID after connecting
      LogicalTransactionId firstLtxid = oconn.getLogicalTransactionId();

      // The subsequent LTXID updates can be obtained via the listener
      oconn.addLogicalTransactionIdEventListener(this);
  }

public class LtxidListenerImpl
  implements LogicalTransactionIdEventListener
{
  ...

  public void onLogicalTransactionIdEvent(LogicalTransactionIdEvent
ltxidEvent)
    {
      LogicalTransactionId newLtxid = ltxidEvent.getLogicalTransactionId();
      // process newLtxid ......
    }
}
```

Using Server-Side Transaction Guard APIs

The DBMS_APP_CONT package contains the GET_LTXID_OUTCOME procedure that contains the server-side Transaction Guard APIs. This procedure forces the outcome of a transaction. If the transaction is not committed, then a fake transaction is committed. Otherwise, the state of the transaction is returned. By default, the EXECUTE privilege for this package is granted to Database Administrators.

Syntax

```
PROCEDURE GET_LTXID_OUTCOME(CLIENT_LTXID        IN  RAW,
```

```
                                committed           OUT BOOLEAN,
                                USER_CALL_COMPLETED OUT BOOLEAN);
```

Input Parameter

CLIENT_LTXID specifies the LTXID from the client driver.

Output Parameter

COMMITTED specifies that the transaction is committed.

USER_CALL_COMPLETED specifies that the user call, which committed the transaction, is complete.

Exceptions

SERVER_AHEAD is thrown when the server is ahead of the client. So, the transaction is an old transaction and must have already been committed.

CLIENT_AHEAD is thrown when the client is ahead of the server. This can only happen if the server is flashed back or the LTXID is corrupted. In either of these situations, the outcome cannot be determined.

ERROR is thrown when an error occurs during processing and the outcome cannot be determined. It specifies the error code raised during the execution of the current procedure.

Example

Example 26–1 shows how you can call the GET_LTXID_OUTCOME procedure and find out the outcome of an LTXID:

Example 26–1 Finding Out the Outcome of an LTXID

```
...
OracleConnection oconn = (OracleConnection) ods.getConnection();
LogicalTransactionId ltxid = oconn.getLogicalTransactionId();
boolean committed = false;
boolean call_completed = false;

try
{
  CallableStatement cstmt = oconn.prepareCall(GET_LTXID_OUTCOME);
  cstmt.setObject(1, ltxid);
  cstmt.registerOutParameter(2, OracleTypes.BIT);
  cstmt.registerOutParameter(3, OracleTypes.BIT);

  cstmt.execute();

  committed = cstmt.getBoolean(2);
  call_completed = cstmt.getBoolean(3);

  System.out.println("LTXID committed ? " + committed);
  System.out.println("User call completed ? " + call_completed);
}
catch (SQLException sqlexc)
{
  System.out.println("Calling GET_LTXID_OUTCOME failed");
  sqlexc.printStackTrace();
}
```

27

Application Continuity for Java

The outages of the underlying software, hardware, communications, and storage layers can cause application execution to fail. In the worst cases, the middle-tier servers may need to be restarted to deal with the logon storms[1]. To overcome such problems, Oracle Database 12*c* Release 1 (12.1) introduces the Application Continuity feature that masks database outages to the application and end users are not exposed to such outages.

> **Note:** Application Continuity is a feature of the Oracle JDBC Thin driver and is not supported by JDBC OCI driver.

Application Continuity provides a general purpose, application-independent solution that enables recovery of work from an application perspective, after the occurrence of a planned or unplanned outage. The outage can be related to system, communication, or hardware following a repair, a configuration change, or a patch application.

See Also:

- *Oracle Database Development Guide* for more information about Application Continuity
- *Oracle Database Development Guide* for more information about Transaction Guard
- "Transaction Guard for Java"

This chapter discusses the JDBC aspect of Application Continuity in the following sections:

- Configuring Oracle JDBC for Application Continuity for Java
- Configuring Oracle Database for Application Continuity for Java
- Identifying Request Boundaries in Application Continuity for Java
- Registering a Connection Initialization Callback in Application Continuity for Java (optional)
- Delaying the Reconnection in Application Continuity for Java
- Retaining Mutable Values in Application Continuity for Java
- Disabling Replay in Application Continuity for Java

[1] "A Logon storm is a sudden increase in the number of client connection requests."

Configuring Oracle JDBC for Application Continuity for Java

You must use either the `oracle.jdbc.replay.OracleDataSourceImpl` or `oracle.jdbc.replay.OracleConnectionPoolDataSourceImpl` data source to obtain JDBC connections. They are new Oracle JDBC data sources, and work similarly to the existing non-XA data sources, such as `oracle.jdbc.pool.OracleDataSource`. You can use both in a standalone manner, or configure them as connection factories for a connection pool, such as Universal Connection Pool (UCP), or Oracle WebLogic Server connection pool.

The following code snippet illustrates their usage in a standalone JDBC application:

```
import java.sql.Connection;
import javax.sql.PooledConnection;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.replay.OracleDataSourceFactory;
import oracle.jdbc.replay.OracleDataSource;
import oracle.jdbc.replay.OracleConnectionPoolDataSource;

...
{
......
OracleDataSource rds = OracleDataSourceFactory.getOracleDataSource();
rds.setUser(user);
rds.setPassword(passwd);
rds.setURL(url);
......  // Other data source configuration like callback, timeouts, etc.

Connection conn = rds.getConnection();
((OracleConnection) conn).beginRequest();  // Explicit request begin
......  // JDBC calls protected by Application Continuity
((OracleConnection) conn).endRequest();  // Explicit request end
conn.close();

OracleConnectionPoolDataSource rcpds =
OracleDataSourceFactory.getOracleConnectionPoolDataSource();
rcpds.setUser(user);
rcpds.setPassword(passwd);
rcpds.setURL(url);
......  // other data source configuration like callback, timeouts, and so on

PooledConnection pc = rcpds.getPooledConnection();
Connection conn2 = pc.getConnection();  // Implicit request begin
......  // JDBC calls protected by Application Continuity
conn2.close();  // Implicit request end
......
```

> **See Also:** "Data Sources and URLs" for more information about Oracle JDBC data sources

You must remember the following points while using the connection URL:

- Always use the thin driver in the connection URL.
- Always connect to a service. Never use `instance_name` or SID because these do not direct to known good instances and SID is deprecated.
- If the addresses in the `ADDRESS_LIST` at the client does not match the `REMOTE_LISTENER` setting for the database, then it does not connect showing `services`

cannot be found. So, the addresses in the ADDRESS_LIST at the client *must* match the REMOTE_LISTENER setting for the database:

- If REMOTE_LISTENER is set to the SCAN_VIP, then the ADDRESS_LIST uses SCAN_VIP
- If REMOTE_LISTENER is set to the host VIPs, then the ADDRESS_LIST uses the same host VIPs
- If REMOTE_LISTENER is set to both SCAN_VIP and host VIPs, then the ADDRESS_LIST uses SCAN_VIP and the same host VIPs

---

Note: For Oracle clients prior to release 11.2, the ADDRESS_LIST must be upgraded to use SCAN, which means expanding the ADDRESS_LIST to three ADDRESS entries corresponding to the three SCAN IP addresses.

If such clients connect to a database that is upgraded from an earlier release through Database Upgrade Assistant, then you must retain the ADDRESS_LIST of these clients set to the HOST VIPs. However, if REMOTE_LISTENER is changed to ONLY SCAN, or the clients are moved to a newly installed Oracle Database 12*c* Release 1, where REMOTE_LISTENER is ONLY SCAN, then they do not get a complete service map, and may not always be able to connect.

- Set RETRY_COUNT, CONNECT_TIMEOUT, and TRANSPORT_CONNECT_TIMEOUT parameters in the connection string. This is a general recommendation for configuring the JDBC thin driver connections, starting from Oracle Database Release 11.2.0.2. These settings improve acquiring new connections at runtime, at replay, and during work drains for planned outages.

  The CONNECT_TIMEOUT parameter is equivalent to the SQLNET.OUTBOUND_CONNECT_TIMEOUT parameter in the sqlnet.ora file and applies to the full connection. The TRANSPORT_CONNECT_TIMEOUT parameter applies as per the ADDRESS parameter. If the service is not registered for a failover or restart, then retrying is important when you use SCAN. For example, for using remote listeners pointing to SCAN addresses, you should use the following settings:

```
jdbc:oracle:thin:@(DESCRIPTION =
 (TRANSPORT_CONNECT_TIMEOUT=3)
 (RETRY_COUNT=20)(FAILOVER=ON)
 (ADDRESS_LIST =(ADDRESS=(PROTOCOL=tcp)
  (HOST=CLOUD-SCANVIP.example.com)(PORT=5221))
 (CONNECT_DATA=(SERVICE_NAME=orcl)))

REMOTE_LISTENERS=CLOUD-SCANVIP.example.com:5221
```

  Similarly, for using remote listeners pointing to VIPs at the database, you should use the following settings:

```
jdbc:oracle:thin:@(DESCRIPTION =
(TRANSPORT_CONNECT_TIMEOUT=3)
(CONNECT_TIMEOUT=60)(RETRY_COUNT=20)(FAILOVER=ON)
 (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=tcp)(HOST=CLOUD-VIP1.example.com)(PORT=5221) )
(ADDRESS=(PROTOCOL=tcp)(HOST=CLOUD-VIP2.example.com)(PORT=5221) )
 (ADDRESS=(PROTOCOL=tcp)(HOST=CLOUD-VIP3.example.com)(PORT=5221) ))
(CONNECT_DATA=(SERVICE_NAME=orcl)))
```

```
REMOTE_LISTENERS=CLOUD-VIP1.example.com:5221
```

See Also:

- *Oracle Database Net Services Reference* for more information about local naming parameters
- *Oracle Real Application Clusters Administration and Deployment Guide*

Configuring Oracle Database for Application Continuity for Java

You must have the following configuration for Oracle Database to use Application Continuity for Java:

- Use Oracle Database 12*c* Release 1 (12.1)
- If you are using Oracle Real Application Clusters (Oracle RAC) or Oracle Data Guard, then ensure that FAN is configured with Oracle Notification System (ONS) to communicate with Oracle WebLogic Server or the Universal Connection Pool (UCP)
- Use an application service for all database work. To create the service you must:
  - Run the `SRVCTL` command if you are using Oracle RAC
  - Use the `DBMS_SERVICE` package if you are not using Oracle RAC
- Set the required properties on the service for replay and load balancing. For example, set:
  - `FAILOVER_TYPE = TRANSACTION` for using Application Continuity
  - `COMMIT_OUTCOME` = `TRUE` for enabling Transaction Guard
  - `REPLAY_INITIATION_TIMEOUT = 900` for setting the duration in seconds for which replay will occur
  - `FAILOVER_RETRIES = 30` for specifying the number of connection retries for each replay
  - `FAILOVER_DELAY = 10` for specifying the delay in seconds between connection retries
  - `GOAL = SERVICE_TIME`, if you are using Oracle RAC, then this is a recommended setting
  - `CLB_GOAL = LONG`, if you are using Oracle RAC, then this is a recommended setting
- Do not use the database service, that is, the default service corresponding to the `DB_NAME` or `DB_UNIQUE_NAME`. This service is reserved for Oracle Enterprise Manager and for DBAs. Oracle does not recommend the use of the database service for high availability because this service cannot be:
  - Enabled and disabled
  - Relocated on Oracle RAC
  - Switched over to Oracle Data Guard

  See Also: *Oracle Database Development Guide* for more information on the operation and usage of Application Continuity.

Identifying Request Boundaries in Application Continuity for Java

A Request is a unit of work on a physical connection to Oracle Database that is protected by Application Continuity. Request demarcation varies with specific use-case scenarios. A request begins when a connection is borrowed from the Universal Connection Pool (UCP) or WebLogic Server connection pool, and ends when this connection is returned to the connection pool.

> **Note:** You cannot borrow a connection from the Database Resident Connection Pool (DRCP) because DRCP does not work with Application Continuity. Refer to Chapter 23, "Database Resident Connection Pooling" for more information about DRCP.

The JDBC driver provides explicit request boundary declaration APIs `beginRequest` and `endRequest` in the `oracle.jdbc.OracleConnection` interface. These APIs enable applications, frameworks, and connection pools to indicate to the JDBC Replay Driver about demarcation points, where it is safe to release the call history, and to enable replay if it had been disabled by a prior request. At the end of the request, the JDBC Replay Driver purges the recorded history on the connection, where the API is called. This helps to further conserve memory consumption for applications that use the same connections for an extended period of time without returning them to the pool.

You do not need to make any changes to your application for identifying request boundaries, if the application uses connections from these Oracle connection pools, or from the `oracle.jdbc.replay.OracleConnectionPoolDataSourceImpl` data source. However, for the connection pool to work, the application must get connections when needed, and release connections when not in use. This scales better and provides request boundaries transparently.

Applications that do not borrow and return connections from the Oracle connection pools should explicitly mark request boundaries. For example, if your application is using custom JDBC pools, then the `beginRequest` method should be called at check-out and the `endRequest` method should be called at check-in. These methods can also be used for standalone JDBC applications without a connection pool.

The APIs have no impact on the applications other than improving resource consumption, recovery, and load balancing performance. These APIs do not involve altering a connection state by calling any JDBC method, SQL, or PL/SQL. An error is returned if an attempt is made to begin or end a request while a local transaction is open.

Registering a Connection Initialization Callback in Application Continuity for Java (optional)

Non-transactional session state (NTSS) is state of a database session that exists outside database transactions and is not protected by recovery. For applications that use stateful requests, the non-transactional state is re-established as the rebuilt session.

For applications that set state only at the beginning of a request, or for stateful applications that gain performance benefits from using connections with a preset state, one among the following callback options are provided:

- No Callback
- Connection Labeling
- Connection Initialization Callback

No Callback

In this scenario, the application builds up its own state during each request.

Connection Labeling

This scenario is applicable only to Universal Connection Pool (UCP) and Oracle WebLogic server. The application can be modified to take advantage of the preset state on connections. Connection Labeling APIs determine how well a connection matches, and use a callback to populate the gap when a connection is borrowed. All applications cannot use Connection Labeling because it requires re-coding to some extent.

See Also: *Oracle Universal Connection Pool for JDBC Developer's Guide*

Connection Initialization Callback

In this scenario, the replay driver uses an application callback to set the initial state of the session during runtime and replay. The JDBC replay driver provides an optional connection initialization callback interface as well as methods for registering and unregistering such callbacks.

When registered, the initialization callback is executed at each successful reconnection following a recoverable error. An application is responsible for ensuring that the initialization actions are the same as that on the original connection before failover. If the callback invocation fails, then replay is disabled on that connection.

This section discusses initialization callbacks in the following sections:

- Creating an Initialization Callback
- Registering an Initialization Callback
- Removing or Unregistering an Initialization Callback

Creating an Initialization Callback

To create a JDBC connection initialization callback, an application implements the `oracle.jdbc.replay.ConnectionInitializationCallback` interface. One callback is allowed for every instance of the `oracle.jdbc.replay.OracleDataSource` interface.

Note: This callback is only invoked during failover, after a successful reconnection.

Example

The following example demonstrates a simple initialization callback implementation:

```
import oracle.jdbc.replay.ConnectionInitializationCallback;
class MyConnectionInitializationCallback implements
ConnectionInitializationCallback
{
    public MyConnectionInitializationCallback()
    {
        ...
    }
    public void initialize(java.sql.Connection connection) throws SQLException
    {
        // Reset the state for the connection, if necessary (like ALTER SESSION)
        ...
```

```
    }
}
```

Registering an Initialization Callback

The JDBC Replay Driver provides the `registerConnectionInitializationCallback(ConnectionInitializationCallback cbk)` method in the `oracle.jdbc.replay.OracleDataSource` interface for registering a connection initialization callback. One callback is allowed for every instance of the `OracleDataSource` interface.

Removing or Unregistering an Initialization Callback

The JDBC Replay Driver provides the `unregisterConnectionInitializationCallback(ConnectionInitializationCallback cbk)` method in the `oracle.jdbc.replay.OracleDataSource` interface for unregistering a connection initialization callback.

Delaying the Reconnection in Application Continuity for Java

By default, when JDBC Replay Driver initiates a failover, the driver attempts to recover the in-flight work at an instance where the service is available. For doing this, the driver must first reestablish a good connection to a working instance. This reconnection can take some time if the database or the instance needs to be restarted before the service is relocated and published. So, the failover should be delayed until the service is available from another instance or database.

You need to use the `FAILOVER_RETRIES` and `FAILOVER_DELAY` parameters to manage reconnecting. These parameters can work well in conjunction with a planned outage, for example, an outage that may make a service unavailable for several minutes. While setting the `FAILOVER_DELAY` and `FAILOVER_RETRIES` parameters, check the value of the `REPLAY_INITIAITION_TIMEOUT` parameter first. The default value for this parameter is 900 seconds. A high value for the `FAILOVER_DELAY` parameter can cause replay to be canceled.

Parameter Name	Possible Value	Default Value
`FAILOVER_RETRIES`	Positive integer zero or above	30
`FAILOVER_DELAY`	Time in seconds	10

Examples

This section provides configuration examples for service creation and modification in the following subsections:

- Creating Services on Oracle RAC
- Modifying Services on Single-Instance Databases

Creating Services on Oracle RAC

If you are using Oracle RAC or Oracle RAC One, then use the `SRVCTL` command to create and modify services in the following way:

- For Policy-Managed Environments
- For Administrator-Managed Environments

For Policy-Managed Environments

```
srvctl add service -d codedb -s GOLD -g Srvpool -j SHORT -B SERVICE_TIME -z 30 -w
10 -commit_outcome
 TRUE -e TRANSACTION -replay_init_time 1800 -retention 86400 -notification TRUE
```

For Administrator-Managed Environments

```
srvctl add service -d codedb -s GOLD -r serv1 -a serv2  -j SHORT -B SERVICE_TIME
-z 30 -w 10 -commit_outcome
 TRUE -e TRANSACTION -replay_init_time 1800 -retention 86400 -notification TRUE
```

Modifying Services on Single-Instance Databases

If you are using a single-instance database, then use the `DBMS_SERVICE` package to modify services in the following way:

```
declare
params dbms_service.svc_parameter_array;
begin
params('FAILOVER_TYPE'):='TRANSACTION';
params('REPLAY_INITIATION_TIMEOUT'):=1800;
params('RETENTION_TIMEOUT'):=604800;
params('FAILOVER_DELAY'):=10;
params('FAILOVER_RETRIES'):=30;
params('commit_outcome'):='true';
params('aq_ha_notifications'):='true';
dbms_service.modify_service('[your service]',params);
end;
/
```

Retaining Mutable Values in Application Continuity for Java

A mutable object is a variable, function return value, or other structure that returns a different value each time that it is called. For example, `Sequence.NextVal`, `SYSDATE`, `SYSTIMESTAMP`, and `SYS_GUID`. To retain the function results for named functions at replay, the DBA must grant `KEEP` privileges to the user who invokes the function. This security restriction is imposed to ensure that it is valid for replay to save and restore function results for code that is not owned by that user.

See Also: *Oracle Database Development Guide*

Grant and Revoke Interface

You can work with mutables values by using the standard `GRANT` and `REVOKE` interfaces in the following way:

- Dates and SYS_GUID Syntax
- Sequence Syntax
- GRANT ALL Statement
- Rules for Grants on Mutable Values

Dates and SYS_GUID Syntax

The `DATE_TIME` and `SYS_GUID` syntax is as follows:

```
GRANT [KEEP DATE_TIME | KEEP SYS_GUID].. [to  USER]
REVOKE [KEEP DATE_TIME | KEEP SYS_GUID] … [from USER]
```

For example, for EBS standard usage with original dates

```
Grant KEEP DATE_TIME,  KEEP SYS_GUID  to [custom user];
Grant KEEP DATE_TIME,  KEEP SYS_GUID  to [apps user];
```

Sequence Syntax

The Sequence syntax can be of the following types:

- Owned Sequence Syntax
- Others Sequence Syntax

Owned Sequence Syntax

```
ALTER SEQUENCE [sequence object] [KEEP|NOKEEP];
```

This command retains the original values of `sequence.nextval` for replaying, so that the keys match after replay. Most applications need to retain the sequence values at replay. The `ALTER SYNTAX` is only for owned sequences.

Others Sequence Syntax

```
GRANT KEEP SEQUENCES.. [to  USER]  on [sequence object];
REVOKE  KEEP SEQUENCES … [from USER]  on [sequence object];
```

For example, use the following command for EBS standard usage with original sequence values:

```
Grant KEEP SEQUENCES  to [apps user] on [sequence object];
Grant KEEP SEQUENCES  to [custom user] on [sequence object];
```

GRANT ALL Statement

The `GRANT ALL` statement grants `KEEP` privilege on all the objects of a user. However, it excludes mutable values, that is, mutable values require explicit grants.

Rules for Grants on Mutable Values

Follow these rules while granting privileges on mutable objects:

- If a user has `KEEP` privilege granted on mutables values, then the objects inherit mutable access when the `SYS_GUID`, `SYSDATE`, and `SYSTIMESTAMP` functions are called.
- If the `KEEP` privilege on mutable values on a sequence object is revoked, then SQL or PL/SQL blocks using that object will not allow mutable collection or application for that sequence.
- If granted privileges are revoked between runtime and failover, then the mutable values that are collected are not applied for replay.
- If new privileges are granted between runtime and failover, mutable values are not collected and these values are not applied for replay.

Disabling Replay in Application Continuity for Java

This section describes the following concepts:

- How to Disable Replay

How to Disable Replay

If any application module uses a design that is unsuitable for replay, then the disable replay API disables replay on a per request basis. Disabling replay can be added to the callback or to the main code by using the `disableReplay` method of the `oracle.jdbc.replay.ReplayableConnection` interface. For example:

```
if (connection instanceof oracle.jdbc.replay.ReplayableConnection)
{
    (( oracle.jdbc.replay.ReplayableConnection)connection).disableReplay();

}
```

Disabling replay does not alter the connection state by reexecuting any JDBC method, SQL or PL/SQL. When replay is disabled using the disable replay API, both recording and replay are disabled until that request ends. There is no API to reenable replay because it is invalid to reestablish the database session with time gaps in a replayed request. This ensures that replay runs *only* if a complete history of needed calls has been recorded.

When to Disable Replay

By default, the JDBC replay driver replays following a recoverable error. The disable replay API can be used in the entry point of application modules that are unable to lose the database sessions and recover. For example, if the application uses the `UTL_SMTP` package and does not want messages to be repeated, then the `disableReplay` API affects only the request that needs to be disabled. All other requests continue to be replayed.

The following are scenarios to consider before configuring an application for replay:

- Application Calls External PL/SQL Actions that Should not Be Repeated
- Application Synchronizes Independent Sessions
- Application Uses Time at the Middle-tier in the Execution Logic
- Application assumes that ROWIds do not change
- Application Assumes that Side Effects Execute Once
- Application Assumes that Location Values Do not Change

Application Calls External PL/SQL Actions that Should not Be Repeated

During replay, autonomous transactions and external PL/SQL calls can have side effects that are separate from the main transaction. These side effects are replayed unless you specify otherwise and leave persistent results behind. These side effects include writing to an external table, sending email, forking sessions out of PL/SQL or Java, transferring files, accessing external URLs, and so on. For example, in case of PL/SQL messaging, suppose, you walk away in-between some work without committing and the session times out. Now, if you issue a `Ctrl+C` command, then the foreground of a component fails. When you resubmit the work, then this side effect can also be repeated.

> **See Also:** *Oracle Database Development Guide* for more information about potential side effects of Application Continuity

You must make a conscious decision about whether to enable replay for external actions or not. For example, you can consider the following situations where this decision is important:

- Using the `UTL_HTTP` package to issue a SOA call
- Using the `UTL_SMTP` package to send a message
- Using the `UTL_URL` package to access a web site

Use the `disableReplay` API if you do not want such external actions to be replayed.

Application Synchronizes Independent Sessions

You can configure an application for replay if the application synchronizes independent sessions using volatile entities that are held until commit, rollback, or session loss. In this case, the application synchronizes multiple sessions connected to several data sources that are otherwise inter-dependent using resources such as a database lock. This synchronization may be fine if the application is only serializing these sessions and understands that any session may fail. However, if the application assumes that a lock or any other volatile resource held by one data source implies exclusive access to data on the same or a separate data source from other connections, then this assumption may be invalidated when replaying.

During replay, the driver is not aware that the sessions are dependent on one session holding a lock or other volatile resource. You can also use pipes, buffered queues, stored procedures taking a resource (such as a semaphore, device, or socket) to implement the synchronization that are lost by failures.

> **Note:** The `DBMS_LOCK` does not replay in the restricted version.

Application Uses Time at the Middle-tier in the Execution Logic

In this case, the application uses the wall clock at the middle-tier as part of the execution logic. The JDBC replay driver does not repeat the middle-tier time logic, but uses the database calls that execute as part of this logic. For example, an application using middle-tier time may assume that a statement executed at Time T1 is not reexecuted at Time T2, unless the application explicitly does so.

Application assumes that ROWIds do not change

If an application caches ROWIDs, then access to these ROWIDs may be invalidated due to database changes. Although a ROWID uniquely identifies a row in a table, a ROWID may change its value in the following situations:

- The underlying table is reorganized
- An index is created on the table
- The underlying table is partitioned
- The underlying table is migrated
- The underlying table is exported and imported using EXP/IMP/DUL
- The underlying table is rebuilt using Golden Gate or Logical Standby or other replication technology
- The database of the underlying table is flashed back or restored

It is bad practice for an application to store ROWIDs for later use as the corresponding row may either not exist or contain completely different data.

Application Assumes that Side Effects Execute Once

In this case, the following are replayed during a replay:

- Autonomous transactions
- Opening of back channels separate to the main transaction side effects

Examples of back channels separate to the main transaction include writing to an external table, sending email, forking sessions out of PL/SQL or Java, writing to output files, transferring files, and writing exception files. Any of these actions leave persistent side effects in the absence of replay. Back channels can leave persistent results behind. For example, if a user leaves a transaction midway without committing and the session times out, then the user presses Ctrl+C, the foreground or any component fails. If the user resubmits work, then the side effects can be repeated.

Application Assumes that Location Values Do not Change

`SYSCONTEXT` options comprise a location-independent set such as National Language Support (NLS) settings, `ISDBA`, `CLIENT_IDENTIFIER`, `MODULE`, and `ACTION`, and a location-dependent set that uses physical locators. Typically, an application does not use the physical identifier, except in testing environments. If physical locators are used in mainline code, then the replay finds the mismatch and rejects it. However, it is fine to use physical locators in callbacks.

Example

```
select
    sys_context('USERENV','DB_NAME')
    ,sys_context('USERENV','HOST')
    ,sys_context('USERENV','INSTANCE')
    ,sys_context('USERENV','IP_ADDRESS')
    ,sys_context('USERENV','ISDBA')
    ,sys_context('USERENV','SESSIONID')
    ,sys_context('USERENV','TERMINAL')
    ,sys_context('USERENV',ID')
from dual
```

Diagnostics and Tracing

The JDBC Replay driver supports standard JDK logging. Logging is enabled using the Java command-line `-Djava.util.logging.config.file=<file>` option. Log level is controlled with the `oracle.jdbc.internal.replay.level` attribute in the log configuration file. For example:

```
oracle.jdbc.internal.replay.level = FINER|FINEST
```

where, `FINER` produces external APIs and `FINEST` produces large volumes of trace. You must use `FINEST` only under supervision.

If you use the `java.util.logging.XMLFormatter` class to format a log record, then the logs are more readable but larger. If you are using replay with FAN enabled on UCP or WebLogic Server, then you should also enable FAN-processing logging.

See Also: *Oracle Universal Connection Pool for JDBC Developer's Guide*

Writing Replay Trace to Console

Following is the example of a configuration file for logging configuration.

```
oracle.jdbc.internal.replay.level = FINER
```

```
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.XMLFormatter
```

Example: Writing Replay Trace to a File

Following is the example of a `properties` file for logging configuration.

```
oracle.jdbc.internal.replay.level = FINEST

# Output File Format (size, number and style)
# count: Number of output files to cycle through, by appending an integer to the
base file name:
# limit: Limiting size of output file in bytes
handlers = java.util.logging.FileHandler
java.util.logging.FileHandler.pattern = [file location]/replay_%U.trc
java.util.logging.FileHandler.limit = 500000000
java.util.logging.FileHandler.count = 1000
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
```

28

Transparent Application Failover

This chapter contains the following sections:

Overview of Transparent Application Failover

Transparent Application Failover (TAF) is a feature of the Java Database Connectivity (JDBC) Oracle Call Interface (OCI) driver. It enables the application to automatically reconnect to a database, if the database instance to which the connection is made fails. In this case, the active transactions roll back.

When an instance to which a connection is established fails or is shut down, the connection on the client-side becomes stale and would throw exceptions to the caller trying to use it. TAF enables the application to transparently reconnect to a preconfigured secondary instance, creating a fresh connection, but identical to the connection that was established on the first original instance. That is, the connection properties are the same as that of the earlier connection. This is true regardless of how the connection was lost.

Note:

- TAF is always active and does not have to be set.
- TAF is not supported with LOB and XML types.

Failover Type Events

The following are possible failover events in the `OracleOCIFailover` interface:

- `FO_SESSION`

  Is equivalent to `FAILOVER_MODE=SESSION` in the `tnsnames.ora` file `CONNECT_DATA` flags. This means that only the user session is authenticated again on the server side, while open cursors in the OCI application need to be reprocessed.

- `FO_SELECT`

Is equivalent to FAILOVER_MODE=SELECT in tnsnames.ora file CONNECT_DATA flags. This means that not only the user session is re-authenticated on the server side, but open cursors in the OCI can continue fetching. This implies that the client-side logic maintains fetch-state of each open cursor.

- FO_NONE

  Is equivalent to FAILOVER_MODE=NONE in the tnsnames.ora file CONNECT_DATA flags. This is the default, in which no failover functionality is used. This can also be explicitly specified to prevent failover from happening. Additionally, FO_TYPE_UNKNOWN implies that a bad failover type was returned from the OCI driver.

- FO_BEGIN

  Indicates that failover has detected a lost connection and failover is starting.

- FO_END

  Indicates successful completion of failover.

- FO_ABORT

  Indicates that failover was unsuccessful and there is no option of retrying.

- FO_REAUTH

  Indicates that a user handle has been re-authenticated.

- FO_ERROR

  Indicates that failover was temporarily unsuccessful, but it gives the application the opportunity to handle the error and retry failover. The usual method of error handling is to issue the sleep method and retry by returning the value FO_RETRY.

- FO_RETRY

  Indicates that the application should retry failover.

- FO_EVENT_UNKNOWN

  Indicates a bad failover event.

TAF Callbacks

TAF callbacks are used in the event of the failure of one database connection, and failover to another database connection. TAF callbacks are callbacks that are registered in case of failover. The callback is called during the failover to notify the JDBC application of events generated. The application also has some control of failover.

Note: The callback setting is optional.

Java TAF Callback Interface

The OracleOCIFailover interface includes the callbackFn method, supporting the following types and events:

```
public interface OracleOCIFailover{

// Possible Failover Types
public static final int FO_SESSION = 1;
public static final int FO_SELECT  = 2;
public static final int FO_NONE  = 3;
public static final int;
```

```
// Possible Failover events registered with callback
public static final int FO_BEGIN   = 1;
public static final int FO_END     = 2;
public static final int FO_ABORT   = 3;
public static final int FO_REAUTH  = 4;
public static final int FO_ERROR  = 5;
public static final int FO_RETRY  = 6;
public static final int FO_EVENT_UNKNOWN = 7;

public int callbackFn (Connection conn,
                       Object ctxt, // ANy thing the user wants to save
                       int type, // One of the possible Failover Types
                       int event ); // One of the possible Failover Events
```

Handling the FO_ERROR Event

In case of an error while failing over to a new connection, the JDBC application is able to retry failover. Typically, the application sleeps for a while and then it retries, either indefinitely or for a limited amount of time, by having the callback return `FO_RETRY`.

Handling the FO_ABORT Event

Callback registered should return the `FO_ABORT` event if the `FO_ERROR` event is passed to it.

Comparison of TAF and Fast Connection Failover

Transparent Application Failover (TAF) differs from Fast Connection Failover in the following ways:

- Application-level connection retries

  TAF supports connection retries only at the OCI/Net layer. Fast Connection Failover supports application-level connection retries. This gives the application control of responding to connection failovers. The application can choose whether to retry the connection or to rethrow the exception.

- Integration with the Universal Connection Pool

  TAF works at the network level on a per-connection basis, which means that the connection cache cannot be notified of failures. Fast Connection Failover is well-integrated with the Universal Connection Pool, which enables the Connection Cache Manager to manage the cache for high availability. For example, failed connections are automatically invalidated in the cache.

- Event-based

  Fast Connection Failover is based on the Oracle RAC event mechanism. This means that Fast Connection Failover is efficient and detects failures quickly for both active and inactive connections.

- Load-balancing support

  Fast Connection Failover supports `UP` event load balancing of connections and run-time work request distribution across active Oracle RAC instances.

See Also: *Oracle Universal Connection Pool for JDBC Developer's Guide*

Note: Oracle recommends *not* to use TAF and Fast Connection Failover in the same application.

29

Single Client Access Name

Single Client Access Name (SCAN) is an Oracle Real Application Clusters (Oracle RAC) feature that provides a single name for clients to access Oracle Databases running in a cluster. This chapter discusses the following concepts related to the SCAN:

Overview of Single Client Access Name

The SCAN is a domain name registered to at least one and up to three IP addresses, either in Domain Naming Service (DNS) or Grid Naming Service (GNS). When you use GNS and Dynamic Host Configuration Protocol (DHCP), Oracle Clusterware configures the Virtual IP (VIP) addresses for the SCAN name that is provided during cluster configuration. The node VIP and the three SCAN VIPs are obtained from the DHCP server when you use GNS.

> **See Also:** *Oracle Clusterware Administration and Deployment Guide* for more information about GNS

If a new server joins the cluster, then Oracle Clusterware dynamically obtains the required VIP address from the DHCP server, updates the cluster resource, and makes the server accessible through GNS. The benefit of using the SCAN is that the connection information of the client does not need to change if you add or remove nodes in the cluster. Having a single name to access the cluster enables the client to use the EZConnect client and the simple JDBC thin URL to access any Database running in the cluster, independent of the active servers in the cluster. The SCAN provides load balancing and failover for client connections to the Database. The SCAN works as a cluster alias for Databases in the cluster.

Configuring the Database Using the SCAN

The SCAN is an essential part of Database configuration. So, by default, the `REMOTE_LISTENER` parameter is set to the SCAN, assuming that the Database is created using standard Oracle tools. This enables the instances to register with the SCAN Listeners as remote listeners to provide information on what services are being provided by the instance, the current load, and a recommendation on how many incoming connections should be directed to the instance.

In this context, you must set the LOCAL_LISTENER parameter to the node-VIP. If you need fully qualified domain names, then ensure that the LOCAL_LISTENER parameter is set to the fully qualified domain name. By default, a node listener is created on each node in the cluster during cluster configuration. With Oracle Grid Infrastructure, the node listener runs out of the Oracle Grid Infrastructure home and listens on the node-VIP using the specified port. The default port is 1521.

Unlike in earlier Database versions, Oracle does not recommend to set your REMOTE_LISTENER parameter to a server side TNSNAMES alias that resolves the host to the SCAN in the address list entry, for example, HOST=sales1-scan. Instead, you must use the simplified SCAN:port syntax as shown in the following table that shows typical setting for a LOCAL_LISTENER and REMOTE_LISTENER:

Name	Type	Value
LOCAL_LISTENER	string	(DESCRIPTION=(ADDRESS_ LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=loca lhost)(PORT=5221))))
REMOTE_LISTENER	string	example.us.oracle.com:5221

Note: If you are using the easy connect naming method, you may need to modify the SQLNET.ORA file to ensure that EZCONNECT is in the list when you specify the order of the naming methods used for the client name resolution lookups.

This section describes the following concepts:

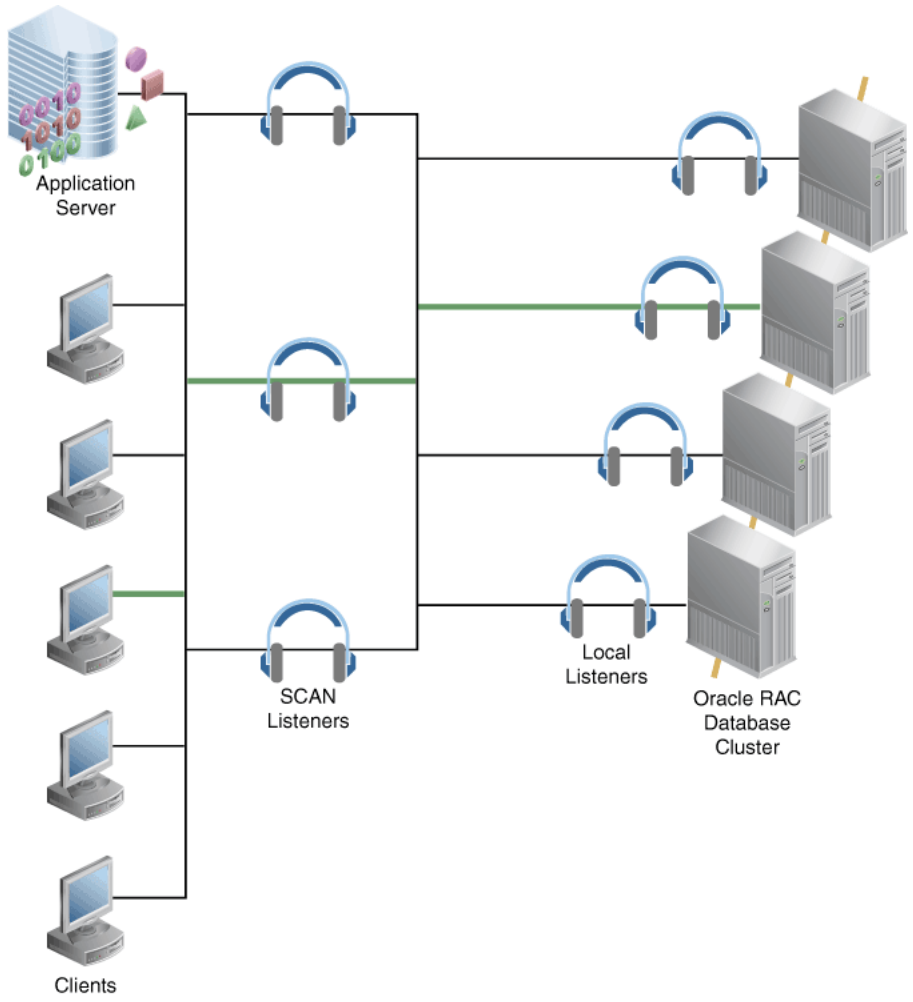
- How Connection Load Balancing Works Using the SCAN
- Version and Backward Compatibility

How Connection Load Balancing Works Using the SCAN

For clients connecting using Oracle SQL*Net, three IP addresses are received by the client by resolving the SCAN name through DNS. The client then goes through the list that it receives from the DNS and tries connecting through one of the IP addresses in the list. If the client receives an error, then it tries connecting to the other addresses before returning an error to the user or application. This is similar to how client connection failover works in earlier Database releases, when an address list is provided in the client connection string.

When a SCAN Listener receives a connection request, the SCAN Listener checks for the least loaded instance providing the requested service. It then re-directs the connection request to the local listener on the node where the least loaded instance is running. Subsequently, the client is given the address of the local listener. The local listener then finally creates the connection to the Database instance.

Figure 29–1 Connection Load Balancing Using the SCAN



Example

This example assumes an Oracle client using a default `TNSNAMES.ora` file:

```
ORCLservice =
(DESCRIPTION =
(ADDRESS = (PROTOCOL = TCP)(HOST = sales1-scan.example.com)(PORT = 1521))
(CONNECT_DATA =
(SERVER = DEDICATED)
(SERVICE_NAME = MyORCLservice)
))
```

Version and Backward Compatibility

To successfully use the SCAN to connect to an Oracle RAC Database in the cluster depends on the following two factors:

- Ability of the client to understand and use the SCAN
- The correct configuration of the `REMOTE_LISTENER` parameter setting in the Database

If the version of the Oracle Client connecting to the Database and the Oracle Database version used are both Oracle Database 11*g* Release 2, and the default configuration is

used as described in the preceding sections, then typically you do not need to make any change to the system.

If both the Oracle Client version and the version of the Oracle Database that the client is connecting to are earlier than Oracle Database 11*g* Release 2, then typically you do not need to make any change to the system. In this case, the client uses a TNS connect descriptor that resolves to the node-VIPs of the cluster, while Oracle Database uses a `REMOTE_LISTENER` entry pointing to the node-VIPs. The disadvantage of this configuration is that the SCAN is not used and therefore every time the cluster changes in the back end, the clients are exposed to changes.

If you are using Oracle Database 11*g* Release 2, but the clients are on an earlier version of the Database, then you must change the Oracle client, or the Oracle Database `REMOTE_LISTENER` parameter settings, or both accordingly. You must consider the following cases in such a scenario:

Table 29–1 Oracle Client and Oracle Database Version Compatibility for the SCAN

Oracle Client Version	Oracle Database Version	Comment
Oracle Database 11*g* Release 2	Oracle Database 11*g* Release 2	No change is required
Oracle Database 11*g* Release 2	Oracle Database Version earlier than Oracle Database 11*g* Release 2	Add the SCAN VIPs as hosts to the `REMOTE_LISTENER` parameter.
Oracle Database Version earlier than Oracle Database 11*g* Release 2	Oracle Database 11*g* Release 2	Update the client `TNSNAMES.ora` file to include the SCAN VIPs. If the Database is upgraded using the Database Upgrade Assistant (DBUA) from a Database earlier than 11*g* Release 2, then the DBUA configures the `REMOTE_LISTENER` parameter to point to the node-VIPs and the SCAN.
Oracle Database Version earlier than Oracle Database 11*g* Release 2	Oracle Database Version earlier than Oracle Database 11*g* Release 2	If you want to make use of the SCAN (recommended), then add the SCAN VIPs as hosts to the `REMOTE_LISTENER` parameter and update the client `TNSNAMES.ora` file to include the SCAN VIPs. Otherwise, no change required.

If you are using a client earlier than Oracle Database 11*g* Release 2, then you cannot fully benefit from the advantages of the SCAN because the Oracle Client cannot handle a set of three IP addresses returned by the DNS for the SCAN. Instead, it tries to connect to only the first address returned in the list and ignores the other two addresses. If the SCAN Listener listening on this specific IP address is not available or the IP address itself is not available, then the connection fails. To ensure load balancing and connection failover with clients earlier than Oracle Database 11*g* Release 2, you must update the `TNSNAMES.ora` file of the client, so that it uses three address lines, where each address line resolves to one of the SCAN VIPs. The following example shows a sample `TNSNAMES.ora` file for a client earlier than Oracle Database 11*g* Release 2:

```
sales.example.com =(DESCRIPTION=
(ADDRESS_LIST= (LOAD_BALANCE=on)(FAILOVER=ON)
(ADDRESS=(PROTOCOL=tcp)(HOST=133.22.67.192)(PORT=1521))
(ADDRESS=(PROTOCOL=tcp)(HOST=133.22.67.193)(PORT=1521))
(ADDRESS=(PROTOCOL=tcp)(HOST=133.22.67.194)(PORT=1521)))
(CONNECT_DATA=(SERVICE_NAME= salesservice.example.com)))
```

Using the SCAN in a Maximum Availability Architecture Environment

If you have a Maximum Availability Architecture (MAA) environment implemented, in which you use Oracle RAC for both your primary and standby Databases that are synchronized using Oracle Data Guard, then using the SCAN provides a simplified TNSNAMES configuration that a client can use to connect to the Database, independent of whether the primary or standby Database is the currently active Database.

To use this simplified configuration, Oracle Database 11*g* Release 2 introduced the following two SQL*Net parameters that can be used for connection strings of individual clients:

- The CONNECT_TIMEOUT parameter

  It specifies the timeout duration in seconds for a client to establish an Oracle Net connection to an Oracle Database. This parameter overrides the SQLNET.OUTBOUT_CONNECT_TIMEOUT parameter in the SQLNET.ORA file.

- The RETRY_COUNT parameter

  It specifies the number of times an ADDRESS_LIST is traversed before the connection attempt is terminated.

Using these two parameters, both the SCANs, the one on the primary site and the one on the standby site, can be used in the client connection strings. Also, if the randomly selected address points to the site that is not currently active, then the timeout enables the connection request to failover before the client waits for an unreasonably long time. The following example shows a sample TNSNAMES.ORA entry for a MAA environment:

```
sales.example.com =(DESCRIPTION= (CONNECT_TIMEOUT=10)(RETRY_COUNT=3)
(ADDRESS_LIST= (LOAD_BALANCE=on)(FAILOVER=ON)
(ADDRESS=(PROTOCOL=tcp)(HOST=sales1-scan)(PORT=1521))
(ADDRESS=(PROTOCOL=tcp)(HOST=sales2-scan)(PORT=1521)))
(CONNECT_DATA=(SERVICE_NAME= salesservice.example.com)))
```

Using the SCAN With Oracle Connection Manager

If you use Oracle Connection Manager (CMAN) with your Oracle RAC Database, then the REMOTE_LISTENER parameter for the Oracle RAC instances must include the CMAN server, so that the CMAN server receives load balancing related information and can load balance connections across the available instances. The easiest way to achieve this is to add the CMAN server as an entry to the REMOTE_LISTENER parameter of the Databases that clients want to connect to through CMAN. You must also remove the SCAN from the TNSNAMES connect descriptor of the clients and configure the CMAN server. The following example shows a server-side TNSNAMES.ora example entry when you use CMAN:

```
SQL> show parameters listener
NAME                 TYPE         VALUE
-------------------------- ----------- ------------------------------
listener_networks    string
local_listener       string       (DESCRIPTION=(ADDRESS_LIST=
                                  (ADDRESS=(PROTOCOL=TCP)
                                  (HOST=148.87.58.109)(PORT=1521))))
remote_listener      string       stscan3.oracle.com:1521,(DESCRIPTION=
                                  (ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
                                  (HOST=CMANserver)(PORT=1521))))
```

See Also: *Oracle Database Net Services Reference* for more information about configuring the CMAN server

Part VII

Transaction Management

This part provides information about transaction management in Oracle Java Database Connectivity (JDBC). It includes a chapter that discusses the Oracle JDBC implementation of distributed transactions.

Part VII contains the following chapter:

30
Distributed Transactions

This chapter discusses the Oracle Java Database Connectivity (JDBC) implementation of distributed transactions. These are multiphased transactions, often using multiple databases, which must be committed in a coordinated way. There is also related discussion of XA, which is a general standard, and not specific to Java, for distributed transactions.

The following topics are discussed:

For further introductory and general information about distributed transactions, refer to the specifications for the JDBC 2.0 Optional Package and the Java Transaction API (JTA).

Overview of Distributed Transactions

A **distributed transaction**, sometimes referred to as a **global transaction**, is a set of two or more related transactions that must be managed in a coordinated way. The transactions that constitute a distributed transaction might be in the same database, but more typically are in different databases and often in different locations. Each individual transaction of a distributed transaction is referred to as a **transaction branch**.

For example, a distributed transaction might consist of money being transferred from an account in one bank to an account in another bank. You would not want either transaction committed without assurance that both will complete successfully.

In JDBC, distributed transaction functionality is built on top of connection pooling functionality. This distributed transaction functionality is also built upon the open XA standard for distributed transactions. XA is part of the X/Open standard and is not specific to Java.

JDBC is used to connect to database resources. However, to include all changes to multiple databases within a transaction, you must use the JDBC connections within a JTA global transaction. The process of including database SQL updates within a transaction is referred to as enlisting a database resource.

The section covers the following topics:

Distributed Transaction Components and Scenarios

In reading the remainder of the distributed transactions section, it will be helpful to keep the following points in mind:

- A distributed transaction system typically relies on an external transaction manager, such as a software component that implements standard JTA functionality, to coordinate the individual transactions.

  Many vendors offer XA-compliant JTA modules, including Oracle, which includes JTA in Oracle9*i* Application Server and Oracle Application Server 10*g*.

- XA functionality is usually isolated from a client application, being implemented instead in a middle-tier environment, such as an application server.

  In many scenarios, the application server and transaction manager will be together on the middle tier, possibly together with some of the application code as well.

- Discussion throughout this section is intended mostly for middle-tier developers.

- The term resource manager is often used in discussing distributed transactions. A resource manager is simply an entity that manages data or some other kind of resource. Wherever the term is used in this chapter, it refers to a database.

Note: Using JTA functionality requires `jta.jar` to be in the `CLASSPATH` environment variable. This file is located at *`ORACLE_HOME`*`/jlib`. Oracle includes this file with the JDBC product.

Distributed Transaction Concepts

When you use XA functionality, the transaction manager uses XA resource instances to prepare and coordinate each transaction branch and then to commit or roll back all transaction branches appropriately.

XA functionality includes the following key components:

- XA data sources

  These are extensions of connection pool data sources and other data sources, and similar in concept and functionality.

  There will be one XA data source instance for each resource manager that will be used in the distributed transaction. You will typically create XA data source instances in your middle-tier software.

  XA data sources produce XA connections.

- XA connections

  These are extensions of pooled connections and similar in concept and functionality. An XA connection encapsulates a physical database connection. Individual connection instances are temporary handles to these physical connections.

An XA connection instance corresponds to a single Oracle session, although the session can be used in sequence by multiple logical connection instances, as with pooled connection instances.

You will typically get an XA connection instance from an XA data source instance in your middle-tier software. You can get multiple XA connection instances from a single XA data source instance if the distributed transaction will involve multiple sessions in the same database.

XA connections produce `OracleXAResource` instances and JDBC connection instances.

- XA resources

  These are used by a transaction manager in coordinating the transaction branches of a distributed transaction.

  You will get one `OracleXAResource` instance from each XA connection instance, typically in your middle-tier software. There is a one-to-one correlation between `OracleXAResource` instances and XA connection instances. Equivalently, there is a one-to-one correlation between `OracleXAResource` instances and Oracle sessions.

  In a typical scenario, the middle-tier component will hand off `OracleXAResource` instances to the transaction manager, for use in coordinating distributed transactions.

  Each `OracleXAResource` instance corresponds to a single Oracle session. So, there can be only a single active transaction branch associated with an `OracleXAResource` instance at any given time. However, there can be additional suspended transaction branches.

  Each `OracleXAResource` instance has the functionality to start, end, prepare, commit, or roll back the operations of the transaction branch running in the session with which the `OracleXAResource` instance is associated.

  The prepare step is the first step of a two-phase commit operation. The transaction manager will issue a `PREPARE` to each `OracleXAResource` instance. Once the transaction manager sees that the operations of each transaction branch have prepared successfully, it will issue a `COMMIT` to each `OracleXAResource` instance to commit all the changes.

- Transaction IDs

  These are used to identify transaction branches. Each ID includes a transaction branch ID component and a distributed transaction ID component. This is how a branch is associated with a distributed transaction. All `OracleXAResource` instances associated with a given distributed transaction would have a transaction ID that includes the same distributed transaction ID component.

- `OracleXAResource.ORATRANSLOOSE`

  Start a loosely coupled transaction with transaction ID `xid`.

Switching Between Global and Local Transactions

Applications can share connections between local and global transactions. Applications can also switch connections between local transactions and global transactions.

A connection is always in one of the following modes:

- `NO_TXN`

No transaction is actively using this connection.

- `LOCAL_TXN`

  A local transaction with auto-commit turned off or disabled is actively using this connection.

- `GLOBAL_TXN`

  A global transaction is actively using this connection.

Each connection switches automatically between these modes depending on the operations carried out on the connection. A connection is always in `NO_TXN` mode when it is instantiated.

> **Note:** The modes are maintained internally by the JDBC drivers in association with Oracle Database.

Table 30–1 describes the connection mode transition rules.

Table 30–1 Connection Mode Transitions

Current Mode	Switches to NO_TXN When	Switches to LOCAL_TXN When	Switches to GLOBAL_TXN When
`NO_TXN`	NA	Auto-commit mode is false and an Oracle data manipulation language (DML) statement is run.	The `start` method is called on an `XAResource` obtained from the `XAconnection` that provided the current connection.
`LOCAL_TXN`	Any of the following happens: ▪ An Oracle data definition language (DDL) statement is run. ▪ `commit` is called. ▪ `rollback` is called, but without parameters.	NA	The `start` method is called on an `XAResource` obtained from the `XAconnection` that provided the current connection. This feature is available starting from Oracle Database 12*c* Release 1 (12.1.0.2).
`GLOBAL_TXN`	Within a global transaction open on this connection, `end` is called on an `XAResource` obtained from the `XAconnection` that provided this connection.	NEVER	NA

If none of these rules is applicable, then the mode does not change.

Mode Restrictions on Operations

The current connection mode restricts which operations are valid within a transaction.

- In the `LOCAL_TXN` mode, applications must not call `prepare`, `commit`, `rollback`, `forget`, or `end` on an `XAResource`. Doing so causes an `XAException` to be thrown.

- In the `GLOBAL_TXN` mode, applications must not call `commit`, `rollback`, `rollback(Savepoint)`, `setAutoCommit(true)`, or `setSavepoint` on a `java.sql.Connection`, and must not call `OracleSetSavepoint` or `oracleRollback` on an `oracle.jdbc.OracleConnection`. Doing so causes a `SQLException` to be thrown.

> **Note:** This mode-restriction error checking is in addition to the standard error checking on the transaction and savepoint APIs.

Oracle XA Packages

Oracle supplies the following three packages that have classes to implement distributed transaction functionality according to the XA standard:

- `oracle.jdbc.xa`
- `oracle.jdbc.xa.client`
- `oracle.jdbc.xa.server`

Classes for XA data sources, XA connections, and XA resources are in both the `client` package and the `server` package. An abstract class for each is in the top-level package. The `OracleXid` and `OracleXAException` classes are in the top-level `oracle.jdbc.xa` package, because their functionality does not depend on where the code is running.

In middle-tier scenarios, you will import `OracleXid`, `OracleXAException`, and the `oracle.jdbc.xa.client` package.

If you intend your XA code to run in the target Oracle Database, however, you will import the `oracle.jdbc.xa.server` package instead of the `client` package.

If code that will run inside a target database must also access remote databases, then do not import either package. Instead, you must fully qualify the names of any classes that you use from the `client` package to access a remote database or from the `server` package to access the local database. Class names are duplicated between these packages.

XA Components

This section discusses the XA components, that is, the standard XA interfaces specified in the JDBC standard, and the Oracle classes that implement them. The following topics are covered:

XADatasource Interface and Oracle Implementation

The `javax.sql.XADataSource` interface outlines standard functionality of XA data sources, which are factories for XA connections. The overloaded `getXAConnection` method returns an XA connection instance and optionally takes a user name and password as input:

```
public interface XADataSource
```

```
{
   XAConnection getXAConnection() throws SQLException;
   XAConnection getXAConnection(String user, String password)
      throws SQLException;
   ...
}
```

Oracle JDBC implements the `XADataSource` interface with the `OracleXADataSource` class, located both in the `oracle.jdbc.xa.client package` and the `oracle.jdbc.xa.server` package.

The `OracleXADataSource` classes also extend the `OracleConnectionPoolDataSource` class, which extends the `OracleDataSource` class, and therefore, include all the connection properties.

The `getXAConnection` methods of the `OracleXADataSource` class returns the Oracle implementation of XA connection instances, which are `OracleXAConnection` instances.

Note: You can register XA data sources in Java Naming Directory and Interface (JNDI) using the same naming conventions as discussed previously for nonpooling data sources.

See Also: For information about Fast Connection Failover, refer to *Oracle Universal Connection Pool for JDBC Developer's Guide*.

XAConnection Interface and Oracle Implementation

An XA connection instance, as with a pooled connection instance, encapsulates a physical connection to a database. This would be the database specified in the connection properties of the XA data source instance that produced the XA connection instance.

Each XA connection instance also has the facility to produce the `OracleXAResource` instance that will correspond to it for use in coordinating the distributed transaction.

An XA connection instance is an instance of a class that implements the standard `javax.sql.XAConnection` interface:

```
public interface XAConnection extends PooledConnection
{
   javax.jta.xa.XAResource getXAResource() throws SQLException;
}
```

As you see, the `XAConnection` interface extends the `javax.sql.PooledConnection` interface, so it also includes the `getConnection`, `close`, `addConnectionEventListener`, and `removeConnectionEventListener` methods.

Oracle JDBC implements the `XAConnection` interface with the `OracleXAConnection` class, located both in the `oracle.jdbc.xa.client package` and the `oracle.jdbc.xa.server` package.

The `OracleXAConnection` classes also extend the `OraclePooledConnection` class.

The `OracleXAConnection` class `getXAResource` method returns the Oracle implementation of an `OracleXAResource` instance, which is an `OracleXAResource` instance. The `getConnection` method returns an `OracleConnection` instance.

A JDBC connection instance returned by an XA connection instance acts as a temporary handle to the physical connection, as opposed to encapsulating the physical

connection. The physical connection is encapsulated by the XA connection instance. The connection obtained from an `XAConnection` object behaves exactly like a regular connection, until it participates in a global transaction. At that time, auto-commit status is set to `false`. After the global transaction ends, auto-commit status is returned to the value it had before the global transaction. The default auto-commit status on a connection obtained from `XAConnection` is `false` in all releases prior to Oracle Database 10*g*. Starting from Oracle Database 10*g*, the default status is `true`.

Each time an XA connection instance `getConnection` method is called, it returns a new connection instance that exhibits the default behavior, and closes any previous connection instance that still exists and had been returned by the same XA connection instance. However, it is advisable to explicitly close any previous connection instance before opening a new one.

Calling the `close` method of an XA connection instance closes the physical connection to the database. This is typically performed in the middle tier.

XAResource Interface and Oracle Implementation

The transaction manager uses `OracleXAResource` instances to coordinate all the transaction branches that constitute a distributed transaction.

Each `OracleXAResource` instance provides the following key functionality, typically invoked by the transaction manager:

- It associates and disassociates distributed transactions with the transaction branch operating in the XA connection instance that produced this `OracleXAResource` instance. Essentially, it associates distributed transactions with the physical connection or session encapsulated by the XA connection instance. This is done through use of transaction IDs.
- It performs the two-phase commit functionality of a distributed transaction to ensure that changes are not committed in one transaction branch before there is assurance that the changes will succeed in all transaction branches.

Note:

- Because there must always be a one-to-one correlation between XA connection instances and `OracleXAResource` instances, an `OracleXAResource` instance is implicitly closed when the associated XA connection instance is closed.
- If a transaction is opened by a given `OracleXAResource` instance, then it must also be closed by the same `OracleXAResource` instance.

An `OracleXAResource` instance is an instance of a class that implements the standard `javax.transaction.xa.XAResource` interface. Oracle JDBC implements the `XAResource` interface with the `OracleXAResource` class, located both in the `oracle.jdbc.xa.client` package and the `oracle.jdbc.xa.server` package.

Oracle JDBC driver creates and returns an `OracleXAResource` instance whenever the `getXAResource` method of the `OracleXAConnection` class is called, and it is Oracle JDBC driver that associates an `OracleXAResource` instance with a connection instance and the transaction branch being run through that connection.

This method is how an `OracleXAResource` instance is associated with a particular connection and with the transaction branch being run in that connection.

OracleXAResource Method Functionality and Input Parameters

The `OracleXAResource` class has several methods to coordinate a transaction branch with the distributed transaction with which it is associated. This functionality usually involves two-phase commit operations.

A transaction manager, receiving `OracleXAResource` instances from a middle-tier component, such as an application server, typically invokes this functionality.

Each of these methods takes a transaction ID as input, in the form of an `Xid` instance, which includes a transaction branch ID component and a distributed transaction ID component. Every transaction branch has a unique transaction ID, but transaction branches belonging to the same global transaction have the same global transaction component as part of their transaction IDs.

start

Starts work on behalf of a transaction branch, associating the transaction branch with a distributed transaction.

```
void start(Xid xid, int flags)
```

The `flags` parameter must be one or more of the following values:

- `XAResource.TMNOFLAGS`

  Flags the start of a new transaction branch for subsequent operations in the session associated with this XA resource instance. This branch will have the transaction ID `xid`, which is an `OracleXid` instance created by the transaction manager. This will map the transaction branch to the appropriate distributed transaction.

- `XAResource.TMJOIN`

  Joins subsequent operations in the session associated with this XA resource instance to the existing transaction branch specified by `xid`.

- `XAResource.TMRESUME`

  Resumes the transaction branch specified by `xid`.

  > **Note:** A transaction branch can be resumed only if it had been suspended earlier.

- `OracleXAResource.TMPROMOTE`

  Promotes a local transaction to a global transaction

- `OracleXAResource.ORATMSERIALIZABLE`

  Starts a serializable transaction with transaction ID `xid`.

- `OracleXAResource.ORATMREADONLY`

  Starts a read-only transaction with transaction ID `xid`.

- `OracleXAResource.ORATMREADWRITE`

  Starts a read/write transaction with transaction ID `xid`.

- `OracleXAResource.ORATRANSLOOSE`

  Starts a loosely coupled transaction with transaction ID `xid`.

`TMNOFLAGS`, `TMJOIN`, `TMRESUME`, `TMPROMOTE`, `ORATMSERIALIZABLE`, `ORATMREADONLY`, and `ORATMREADWRITE` are defined as `static` members of the `XAResource` interface and

OracleXAResource class. ORATMSERIALIZABLE, ORATMREADONLY, and ORATMREADWRITE are the isolation-mode flags. The default isolation behavior is READ COMMITTED.

Note:

- Instead of using the start method with TMRESUME, the transaction manager can cast to OracleXAResource and use the resume(Xid xid) method, an Oracle extension.
- If you use TMRESUME, then you must also use TMNOMIGRATE, as in start(xid, XAResource.TMRESUME | OracleXAResource.TMNOMIGRATE). This prevents the application from receiving the error ORA 1002: fetch out of sequence.
- If you use the isolation-mode flags incorrectly, then an exception with code XAER_INVAL is raised. Furthermore, you cannot use isolation-mode flags when resuming a global transaction, because you cannot set the isolation level of an existing transaction. If you try to use the isolation-mode flags when resuming a transaction, then an external Oracle exception with code ORA-24790 is raised.
- In order to avoid Error ORA 1002: fetch out of sequence, include the TMNOMIGRATE flag as part of the start method. For example:

  ```
  start(xid, XAResource.TMSUSPEND |
  OracleXAResource.TMNOMIGRATE);
  ```

- All the flags defined in OracleXAResource are Oracle extensions. When writing a transaction manager that uses these flags, you should be mindful of this.

Note that to create an appropriate transaction ID in starting a transaction branch, the transaction manager must know to which distributed transaction the transaction branch belongs. The mechanics of this are handled between the middle tier and transaction manager.

end

Ends work on behalf of the transaction branch specified by xid, disassociating the transaction branch from its distributed transaction.

```
void end(Xid xid, int flags)
```

The flags parameter can have one of the following values:

- XAResource.TMSUCCESS

  This is to indicate that this transaction branch is known to have succeeded.

- XAResource.TMFAIL

  This is to indicate that this transaction branch is known to have failed.

- XAResource.TMSUSPEND

  This is to suspend the transaction branch specified by xid. By suspending transaction branches, you can have multiple transaction branches in a single

session. Only one can be active at any given time, however. Also, this tends to be more expensive in terms of resources than having two sessions.

`TMSUCCESS`, `TMFAIL`, and `TMSUSPEND` are defined as static members of the `XAResource` interface and `OracleXAResource` class.

Note:

- Instead of using the `end` method with `TMSUSPEND`, the transaction manager can cast to `OracleXAResource` and use the `suspend(Xid xid)` method, an Oracle extension.
- This XA functionality to suspend a transaction provides a way to switch between various transactions within a single JDBC connection. You can use the XA classes to accomplish this, even if you are not in a distributed transaction environment and would otherwise have no need for the XA classes.
- If you use `TMSUSPEND`, then you must also use `TMNOMIGRATE`, as in `end(xid, XAResource.TMSUSPEND | OracleXAResource.TMNOMIGRATE)`. This prevents the application from receiving the error `ORA 1002: fetch out of sequence`.
- In order to avoid `Error ORA 1002: fetch out of sequence`, include the `TMNOMIGRATE` flag as part of the end method. For example:

  ```
  end(xid, XAResource.TMSUSPEND |
  OracleXAResource.TMNOMIGRATE);
  ```

- All the flags defined in `OracleXAResource` are Oracle extensions. Any transaction manager that uses these flags should take heed of this.

---

prepare

Prepares the changes performed in the transaction branch specified by `xid`. This is the first phase of a two-phase commit operation, to ensure that the database is accessible and that the changes can be committed successfully.

```
int prepare(Xid xid)
```

This method returns an integer value as follows:

- `XAResource.XA_RDONLY`

  This is returned if the transaction branch runs only read-only operations such as `SELECT` statements.

- `XAResource.XA_OK`

  This is returned if the transaction branch runs updates that are all prepared without error.

- NA (no value returned)

  No value is returned if the transaction branch runs updates and any of them encounters errors during preparation. In this case, an XA exception is thrown.

`XA_RDONLY` and `XA_OK` are defined as `static` members of the `XAResource` interface and `OracleXAResource` class.

Note:

- Always call the `end` method on a branch before calling the `prepare` method.
- If there is only one transaction branch in a distributed transaction, then there is no need to call the `prepare` method. You can call the `OracleXAResource` `commit` method without preparing first.

commit

Commits prepared changes in the transaction branch specified by `xid`. This is the second phase of a two-phase commit and is performed only after all transaction branches have been successfully prepared.

```
void commit(Xid xid, boolean onePhase)
```

Set the `onePhase` parameter as follows:

- `true`

  This is to use one-phase instead of two-phase protocol in committing the transaction branch. This is appropriate if there is only one transaction branch in the distributed transaction; the `prepare` step would be skipped.

- `false`

  This is to use two-phase protocol in committing the transaction branch.

rollback

Rolls back prepared changes in the transaction branch specified by `xid`.

```
void rollback(Xid xid)
```

forget

Tells the resource manager to forget about a heuristically completed transaction branch.

```
public void forget(Xid xid)
```

recover

The transaction manager calls this method during recovery to obtain the list of transaction branches that are currently in prepared or heuristically completed states.

```
public Xid[] recover(int flag)
```

Note: Values for `flag` other than `TMSTARTRSCAN`, `TMENDRSCAN`, or `TMNOFLAGS`, cause an exception to be thrown, otherwise `flag` is ignored.

The resource manager returns zero or more `Xid`s for the transaction branches that are currently in a prepared or heuristically completed state. If an error occurs during the operation, then the resource manager throws the appropriate `XAException`.

Note: The `recover` method requires SELECT privilege on `DBA_PENDING_TRANSACTIONS` and EXECUTE privilege on `SYS.DBMS_XA` in Oracle database server. For database versions prior to Oracle Database 11*g* Release 1, where an Oracle patch including a fix for bug 5945463 is not available, or it is infeasible to apply the patch for the particular application scenario, the `recover` method requires `SYSBDBA` privilege. Regular use of `SYSDBA` privilege is a security risk. So, Oracle strongly recommends that you upgrade your Database or apply the fix for bug 5945463, if you need to use the `recover` method.

isSameRM

To determine if two `OracleXAResource` instances correspond to the same resource manager, call the `isSameRM` method from one `OracleXAResource` instance, specifying the other `OracleXAResource` instance as input. In the following example, presume `xares1` and `xares2` are `OracleXAResource` instances:

```
boolean sameRM = xares1.isSameRM(xares2);
```

Xid Interface and Oracle Implementation

The transaction manager creates transaction ID instances and uses them in coordinating the branches of a distributed transaction. Each transaction branch is assigned a unique transaction ID, which includes the following information:

- Format identifier

  A format identifier specifies a Java transaction manager. For example, there could be a format identifier `orcl`. This field *cannot* be null. The size of a format identifier is 4 bytes.

- Global transaction identifier

  It is also known as a distributed transaction ID component. The size of a global transaction identifier is 64 bytes.

- Branch qualifier

  It is also known as transaction branch ID component. The size of a branch qualifier is 64 bytes.

The 64-byte global transaction identifier value will be identical in the transaction IDs of all transaction branches belonging to the same distributed transaction. However, the overall transaction ID is unique for every transaction branch.

An XA transaction ID instance is an instance of a class that implements the standard `javax.transaction.xa.Xid` interface, which is a Java mapping of the X/Open transaction identifier XID structure.

Oracle implements this interface with the `OracleXid` class in the `oracle.jdbc.xa` package. `OracleXid` instances are employed only in a transaction manager, transparent to application programs or an application server.

Note: Oracle does not require the use of `OracleXid` for `OracleXAResource` calls. Instead, use any class that implements the `javax.transaction.xa.Xid` interface.

A transaction manager may use the following in creating an `OracleXid` instance:

```
public OracleXid(int fId, byte gId[], byte bId[]) throws XAException
```

`fId` is an integer value for the format identifier, `gId[]` is a byte array for the global transaction identifier, and `bId[]` is a byte array for the branch qualifier.

The `Xid` interface specifies the following getter methods:

- `public int getFormatId()`
- `public byte[] getGlobalTransactionId()`
- `public type[] getBranchQualifier()`

Error Handling and Optimizations

This section focuses on the functionality of XA exceptions and error handling and the Oracle optimizations in its XA implementation. It covers the following topics:

- XAException Classes and Methods
- Mapping Between Oracle Errors and XA Errors
- XA Error Handling
- Oracle XA Optimizations

The exception and error-handling discussion includes the standard XA exception class and the Oracle-specific XA exception class, as well as particular XA error codes and error-handling techniques.

XAException Classes and Methods

XA methods throw XA exceptions, as opposed to general exceptions or `SQLExceptions`. An XA exception is an instance of the standard class `javax.transaction.xa.XAException` or a subclass.

An Oracle XAException is an instance that consists of an Oracle error portion and an XA error portion. Oracle provides the `oracle.jdbc.xa.OracleXAException` subclasses of the standard `javax.transaction.xa.XAException` class. An `OracleXAException` instance is constructed using one of the following constructors:

```
public OracleXAException()

public OracleXAException(int error)
```

The error value is an error code that combines an Oracle SQL error value and an XA error value. The JDBC driver determines exactly how to combine the Oracle and XA error values.

The `OracleXAException` class has the following methods:

- `public int getOracleError()`

  This method returns the Oracle SQL error code pertaining to the exception, a standard ORA error number or 0 if there is no Oracle SQL error.

- `public int getXAError()`

  This method returns the XA error code pertaining to the exception. XA error values are defined in the `javax.transaction.xa.XAException` class.

Mapping Between Oracle Errors and XA Errors

Oracle errors correspond to XA errors in `OracleXAException` instances as documented in Table 30–2.

Table 30–2 Oracle-XA Error Mapping

Oracle Error Code	XA Error Code
ORA 24756	XAException.XAER_NOTA
ORA 24764	XAException.XA_HEURCOM
ORA 24765	XAException.XA_HEURRB
ORA 24766	XAException.XA_HEURMIX
ORA 24767	XAException.XA_RDONLY
ORA 25351	XAException.XA_RETRY
ORA 30006	XAException.XA_RETRY
ORA 24763	XAException.XAER_PROTO
ORA 24769	XAException.XAER_PROTO
ORA 24770	XAException.XAER_PROTO
ORA 24776	XAException.XAER_PROTO
ORA 2056	XAException.XAER_PROTO
ORA 17448	XAException.XAER_PROTO
ORA 24768	XAException.XAER_PROTO
ORA 24775	XAException.XAER_PROTO
ORA 24761	XAException.XA_RBROLLBACK
ORA 2091	XAException.XA_RBROLLBACK
ORA 2092	XAException.XA_RBROLLBACK
ORA 24780	XAException.XAER_RMERR
All other ORA errors	XAException.XAER_RMFAIL

XA Error Handling

The following example uses the `OracleXAException` class to process an XA exception:

```
try {
   ...
   ...Perform XA operations...
   ...
} catch(OracleXAException oxae) {
  int oraerr = oxae.getOracleError();
  System.out.println("Error " + oraerr);
}
  catch(XAException xae)
{...Process generic XA exception...}
```

In case the XA operations did not throw an Oracle-specific XA exception, the code drops through to process a generic XA exception.

Oracle XA Optimizations

Oracle JDBC has functionality to improve performance if two or more branches of a distributed transaction use the same database instance, meaning that the `OracleXAResource` instances associated with these branches are associated with the same resource manager.

In such a circumstance, the `prepare` method of only one of these `OracleXAResource` instances will return `XA_OK` or will fail. The rest will return `XA_RDONLY`, even if updates are made. This allows the transaction manager to implicitly join all the transaction branches and commit or roll back, in case of failure, the joined transaction through the `OracleXAResource` instance that returned `XA_OK` or failed.

The transaction manager can use the `OracleXAResource` class `isSameRM` method to determine if two `OracleXAResource` instances are using the same resource manager. This way it can interpret the meaning of `XA_RDONLY` return values.

Implementing a Distributed Transaction

This section provides an example of how to implement a distributed transaction using Oracle XA functionality. This section covers the following topics:

- Summary of Imports for Oracle XA
- Oracle XA Code Sample

Summary of Imports for Oracle XA

You must import the following for Oracle XA functionality:

```
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;
```

The `oracle.jdbc.pool` package has classes for connection pooling functionality, some of which have XA-related classes as subclasses.

Alternatively, if the code will run inside Oracle Database and access that database for SQL operations, you must import `oracle.jdbc.xa.server` instead of `oracle.jdbc.xa.client`.

```
import oracle.jdbc.xa.server.*;
```

If your application must access another Oracle Database as part of an XA transaction using the server-side Thin driver, then your code can use the fully qualified names of the `oracle.xa.client` classes.

The `client` and `server` packages each have versions of the `OracleXADataSource`, `OracleXAConnection`, and `OracleXAResource` classes. Abstract versions of these three classes are in the top-level `oracle.jdbc.xa` package.

Oracle XA Code Sample

This example uses a two-phase distributed transaction with two transaction branches, each to a separate database.

Note that for simplicity, this example combines code that would typically be in a middle tier with code that would typically be in a transaction manager, such as the `OracleXAResource` method invocations and the creation of transaction IDs.

For brevity, the specifics of creating transaction IDs and performing SQL operations are not shown here. The complete example is shipped with the product.

This example performs the following sequence:

1. Start transaction branch #1.
2. Start transaction branch #2.
3. Execute DML operations on branch #1.
4. Execute DML operations on branch #2.
5. End transaction branch #1.
6. End transaction branch #2.
7. Prepare branch #1.
8. Prepare branch #2.
9. Commit branch #1.
10. Commit branch #2.

```
// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;

class XA4
{
  public static void main (String args [])
       throws SQLException
  {

    try
    {
        String URL1 = "jdbc:oracle:oci:@";
        // You can put a database name after the @ sign in the connection URL.
        String URL2 ="jdbc:oracle:thin:@(description=(address=(host=localhost)
                     (protocol=tcp)(port=5521))(connect_data=(service_
name=orcl)))";
        // Create first DataSource and get connection
        OracleDataSource ods1 = new OracleDataSource();
        ods1.setURL(URL1);
        ods1.setUser("HR");
        ods1.setPassword("hr");
        Connection conna = ods1.getConnection();

        // Create second DataSource and get connection
        OracleDataSource ods2 = new OracleDataSource();
        ods2.setURL(URL2);
        ods2.setUser("HR");
        ods2.setPassword("hr");
        Connection connb = ods2.getConnection();

        // Prepare a statement to create the table
        Statement stmta = conna.createStatement ();
```

```
        // Prepare a statement to create the table
        Statement stmtb = connb.createStatement ();

        try
        {
          // Drop the test table
          stmta.execute ("drop table my_table");
        }
        catch (SQLException e)
        {
          // Ignore an error here
        }

        try
        {
          // Create a test table
          stmta.execute ("create table my_table (col1 int)");
        }
        catch (SQLException e)
        {
          // Ignore an error here too
        }

        try
        {
          // Drop the test table
          stmtb.execute ("drop table my_tab");
        }
        catch (SQLException e)
        {
          // Ignore an error here
        }

        try
        {
          // Create a test table
          stmtb.execute ("create table my_tab (col1 char(30))");
        }
        catch (SQLException e)
        {
          // Ignore an error here too
        }

        // Create XADataSource instances and set properties.
        OracleXADataSource oxds1 = new OracleXADataSource();
        oxds1.setURL("jdbc:oracle:oci:@");
        oxds1.setUser("HR");
        oxds1.setPassword("hr");

        OracleXADataSource oxds2 = new OracleXADataSource();

        oxds2.setURL("jdbc:oracle:thin:@(description=(address=(host=localhost)
                    (protocol=tcp)(port=5521))(connect_data=(service_
name=orcl)))");
        oxds2.setUser("HR");
        oxds2.setPassword("hr");

        // Get XA connections to the underlying data sources
        XAConnection pc1  = oxds1.getXAConnection();
        XAConnection pc2  = oxds2.getXAConnection();
```

```
// Get the physical connections
Connection conn1 = pc1.getConnection();
Connection conn2 = pc2.getConnection();

// Get the XA resources
XAResource oxar1 = pc1.getXAResource();
XAResource oxar2 = pc2.getXAResource();

// Create the Xids With the Same Global Ids
Xid xid1 = createXid(1);
Xid xid2 = createXid(2);

// Start the Resources
oxar1.start (xid1, XAResource.TMNOFLAGS);
oxar2.start (xid2, XAResource.TMNOFLAGS);

// Execute SQL operations with conn1 and conn2
doSomeWork1 (conn1);
doSomeWork2 (conn2);

// END both the branches -- IMPORTANT
oxar1.end(xid1, XAResource.TMSUCCESS);
oxar2.end(xid2, XAResource.TMSUCCESS);

// Prepare the RMs
int prp1 =  oxar1.prepare (xid1);
int prp2 =  oxar2.prepare (xid2);

System.out.println("Return value of prepare 1 is " + prp1);
System.out.println("Return value of prepare 2 is " + prp2);

boolean do_commit = true;

if (!((prp1 == XAResource.XA_OK) || (prp1 == XAResource.XA_RDONLY)))
   do_commit = false;

if (!((prp2 == XAResource.XA_OK) || (prp2 == XAResource.XA_RDONLY)))
   do_commit = false;

System.out.println("do_commit is " + do_commit);
System.out.println("Is oxar1 same as oxar2 ? " + oxar1.isSameRM(oxar2));

if (prp1 == XAResource.XA_OK)
  if (do_commit)
     oxar1.commit (xid1, false);
  else
     oxar1.rollback (xid1);

if (prp2 == XAResource.XA_OK)
  if (do_commit)
     oxar2.commit (xid2, false);
  else
     oxar2.rollback (xid2);

 // Close connections
conn1.close();
conn1 = null;
conn2.close();
conn2 = null;
```

```
        pc1.close();
        pc1 = null;
        pc2.close();
        pc2 = null;

        ResultSet rset = stmta.executeQuery ("select col1 from my_table");
        while (rset.next())
          System.out.println("Col1 is " + rset.getInt(1));

        rset.close();
        rset = null;

        rset = stmtb.executeQuery ("select col1 from my_tab");
        while (rset.next())
          System.out.println("Col1 is " + rset.getString(1));

        rset.close();
        rset = null;

        stmta.close();
        stmta = null;
        stmtb.close();
        stmtb = null;

        conna.close();
        conna = null;
        connb.close();
        connb = null;

      } catch (SQLException sqe)
      {
        sqe.printStackTrace();
      } catch (XAException xae)
      {
        if (xae instanceof OracleXAException) {
          System.out.println("XA Error is " +
                       ((OracleXAException)xae).getXAError());
          System.out.println("SQL Error is " +
                       ((OracleXAException)xae).getOracleError());
        }
      }
    }

    static Xid createXid(int bids)
      throws XAException
    {...Create transaction IDs...}

    private static void doSomeWork1 (Connection conn)
     throws SQLException
    {...Execute SQL operations...}

    private static void doSomeWork2 (Connection conn)
     throws SQLException
    {...Execute SQL operations...}
}
```

Native-XA in Oracle JDBC Drivers

In general, XA commands can be sent to the server in the following ways:

- Through non-native APIs
- Through native APIs

There is a huge performance difference between the two methods of sending XA commands to the server. The use of native APIs provide high performance gains as compared to the use of non-native APIs.

Prior to Oracle Database 10*g*, the Thin driver used non-native APIs to send XA commands to the server because Thin native APIs were not available. The non-native APIs use PL/SQL procedures, so they have the following disadvantages:

- They require different messages on the wire.
- They cause more round-trips to the database.
- They cause more cursors to remain open.
- They damage statement caching by occupying space in the Statement Cache.

Moreover, the implementation of non-native APIs is in the server. So, in order to solve any problem in sending XA commands, it requires a server patch. This creates a major issue because sometimes the patch requires restarting the server.

Starting from Oracle Database 10*g*, the Thin native APIs are available and are used to send XA commands, by default. Native APIs are more than 10 times faster than the non-native ones.

This section covers the following topics:

- OCI Native XA
- Thin Native XA

OCI Native XA

Native XA is enabled through the use of the `tnsEntry` and `nativeXA` properties of the `OracleXADataSource` class.

> **See Also:** Table 8–2, " Oracle Extended Data Source Properties" on page 8-3 for explanation of these properties.

Note: Currently, OCI Native XA does not work in a multithreaded environment. The OCI driver uses the C/XA library of Oracle to support distributed transactions, which requires that an XAConnection be obtained for each thread before resuming a global transaction.

Configuration and Installation

On a Solaris or Linux system, you need the `libheteroxa11.so` shared library to enable the Native XA feature. This library must be installed and available in the search path for the Native XA feature to work properly.

On a Microsoft Windows system, you need the `heteroxa11.dll` file to enable the Native XA feature. This file must be installed and available in the Windows DLL path for the Native XA feature to work properly.

Exception Handling

When using the Native XA feature in distributed transactions, it is recommended that the application simply check for `XAException` or `SQLException`, rather than `OracleXAException` or `OracleSQLException`.

> **See Also:** "Native XA Messages" on page D-12 for a listing of Native XA messages.

> **Note:** The mapping from SQL error codes to standard XA error codes does not apply to the Native XA feature.

Native XA Code Example

The following portion of code shows how to enable the Native XA feature:

```
...
// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL(url);

// Set the nativeXA property to use Native XA feature
oxds.setNativeXA(true);

// Set the tnsEntry property to an older DB as required
oxds.setTNSEntryName("ora805");
...
```

Thin Native XA

Like the JDBC OCI driver, the JDBC Thin driver also provides support for Native XA. However, the JDBC Thin driver provides support for Native XA by default. This is unlike the case of the JDBC OCI driver in which the support for Native XA is not enabled by default.

You can disable Native XA by calling `setNativeXA(false)` on the XA data source as follows:

```
...
// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
...
// Disabling Native XA
oxds.setNativeXA(false);
...
```

For example, you may need to disable Native XA as a workaround for a bug in the Native XA code.

Part VIII

Manageability

This part discusses the database management and diagnosability support in Oracle Java Database Connectivity (JDBC) drivers.

Part VIII contains the following chapters:

31
Database Administration

This chapter discusses the database administration methods introduced in Oracle Database 11*g* Release 1. This chapter contains the following sections:

Using the Database Administration Methods

Starting from Oracle Database 11*g* Release 1, two JDBC methods, `startup` and `shutdown`, has been added in the `oracle.jdbc.OracleConnection` interface, which enable you to start up and shut down an Oracle Database instance. This is similar to the way you would start up or shut down a database instance from SQL*Plus.

To use the `startup` and `shutdown` methods, you must:

- Have a dedicated connection to the server. You cannot be connected to a shared server through a dispatcher.
- Be connected as `SYSDBA` or `SYSOPER`. To connect as `SYSDBA` or `SYSOPER` with Oracle JDBC drivers, you need to set the `INTERNAL_LOGON` connection property accordingly.

To log on as `SYSDBA` with the JDBC Thin driver you must configure the server to use the password file. For example, to configure `system/manager` to connect as `SYSDBA` with the JDBC Thin driver, perform the following:

1. From the command line, type:

```
orapwd file=$ORACLE_HOME/dbs/orapw entries=5
Enter password: password
```

2. Connect to database as `SYSDBA` and run the following commands from SQL*Plus:

```
GRANT SYSDBA TO system;
PASSWORD system
       Changing password for system
       New password: password
       Retype new password: password
```

3. Edit `init.ora` and add the following line:

```
REMOTE_LOGIN_PASSWORDFILE=EXCLUSIVE
```

4. Restart the database instance.

As opposed to the JDBC Thin driver, the JDBC OCI driver can connect as SYSDBA or SYSOPER locally without specifying a password file on the server.

Using the startup Method

To start a database instance using the startup method, the application must first connect to the database as a SYSDBA or SYSOPER in the PRELIM_AUTH mode, which is the only connection mode that is permitted when the database is down. You can do this by setting the connection property PRELIM_AUTH to true. In the PRELIM_AUTH mode, you can *only* start up a database instance that is down. You *cannot* run any SQL statements in this mode.

Example

The following code snippet shows how to start up a database instance that is down:

```
 OracleDataSource ds = new OracleDataSource();
    Properties prop = new Properties();
    prop.setProperty("user","sys");
    prop.setProperty("password","manager");
    prop.setProperty("internal_logon","sysdba");
    prop.setProperty("prelim_auth","true");
    ds.setConnectionProperties(prop);

ds.setURL("jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=XYZ.com)(PO
RT=5221))"
+ "(CONNECT_DATA=(SERVICE_NAME=rdbms.devplmt.XYZ.com)))");
    OracleConnection conn = (OracleConnection)ds.getConnection();
    conn.startup(OracleConnection.DatabaseStartupMode.NO_RESTRICTION);
    conn.close();
```

Note: The startup method will start up the database using the server parameter file. Oracle JDBC drivers do *not* support database startup using the client parameter file.

Database Startup Options

The startup method takes a parameter that specifies the database startup option. Table 31–1 lists the supported database startup options. These options are defined in the oracle.jdbc.OracleConnection.DatabaseStartupMode class.

Table 31–1 Supported Database Startup Options

Option	Description
FORCE	Shuts down the current instance, if any, of database in the abort mode before starting a new instance.
NO_RESTRICTION	Starts up the database with no restrictions.
RESTRICT	Starts up the database and allows database access only to users with both the CREATE SESSION and RESTRICTED SESSION privileges, typically, the DBA.

The startup method only starts up a database instance. It neither mounts it nor opens it. For mounting and opening the database instance, you have to reconnect as SYSDBA or SYSOPER, without the PRELIM_AUTH mode.

Example

The following code snippet shows how to mount and open a database instance:

```
OracleDataSource ds1 = new OracleDataSource();
Properties prop1 = new Properties();
prop1.setProperty("user","sys");
prop1.setProperty("password","manager");
prop1.setProperty("internal_logon","sysdba");
ds1.setConnectionProperties(prop1);
ds1.setURL(DB_URL);
OracleConnection conn1 = (OracleConnection)ds1.getConnection();
Statement stmt = conn1.createStatement();
stmt.executeUpdate("ALTER DATABASE MOUNT");
stmt.executeUpdate("ALTER DATABASE OPEN");
```

Using the shutdown Method

The `shutdown` method enables you to shut down an Oracle Database instance. To use this method, you must be connected to the database as a `SYSDBA` or `SYSOPER`.

Example

The following code snippet shows how to shut down a database instance:

```
    OracleDataSource ds2 = new OracleDataSource();
...
    OracleConnection conn2 = (OracleConnection)ds2.getConnection();
    conn2.shutdown(OracleConnection.DatabaseShutdownMode.IMMEDIATE);
    Statement stmt1 = conn2.createStatement();
    stmt1.executeUpdate("ALTER DATABASE CLOSE NORMAL");
    stmt1.executeUpdate("ALTER DATABASE DISMOUNT");
    stmt1.close();
    conn2.shutdown(OracleConnection.DatabaseShutdownMode.FINAL);
    conn2.close();
```

Database Startup Options

Like the `startup` method, the `shutdown` method also takes a parameter. In this case, the parameter specifies the database shutdown option. Table 31–2 lists the supported database shutdown options. These options are defined in the `oracle.jdbc.OracleConnection.DatabaseShutdownMode` class.

Table 31–2 Supported Database Shutdown Options

Option	Description
`ABORT`	Does not wait for current calls to complete or users to disconnect from the database.
`CONNECT`	Refuses any new connection and waits for existing connection to end.
`FINAL`	Shuts down the database.
`IMMEDIATE`	Does not wait for current calls to complete or users to disconnect from the database.
`TRANSACTIONAL`	Refuses new transactions and waits for active transactions to end.
`TRANSACTIONAL_LOCAL`	Refuses new local transactions and waits for active local transactions to end.

For shutdown options other than `ABORT` and `FINAL`, you must call the `shutdown` method again with the `FINAL` option to actually shut down the database.

Note: The `shutdown(DatabaseShutdownMode.FINAL)` method must be preceded by another call to the `shutdown` method with one of the following options: `CONNECT`, `TRANSACTIONAL`, `TRANSACTIONAL_LOCAL`, or `IMMEDIATE`. Otherwise, the call hangs.

Standard Database Shutdown Process

A standard way to shut down the database is as follows:

1. Initiate shutdown by prohibiting further connections or transactions in the database. The shut down option can be either `CONNECT`, `TRANSACTIONAL`, `TRANSACTIONAL_LOCAL`, or `IMMEDIATE`.
2. Dismount and close the database by calling the appropriate `ALTER DATABASE` command.
3. Finish shutdown using the `FINAL` option.

In special circumstances to shut down the database as fast as possible, the `ABORT` option can be used. This is the equivalent to `SHUTDOWN ABORT` in SQL*Plus.

A Complete Example

Example 31–1 illustrates the use of the `startup` and `shutdown` methods.

Example 31–1 Database Startup and Shutdown

```
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
/**
 * To logon as sysdba, you need to create a password file for user "sys":
 *   orapwd file=/path/orapw password=password entries=300
 * and add the following setting in init.ora:
 *   REMOTE_LOGIN_PASSWORDFILE=EXCLUSIVE
 * then restart the database.
 */
public class DBStartup
{
  static final String DB_URL =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=localhost)(PORT=5221)
)"
+ "(CONNECT_DATA=(SERVICE_NAME=rdbms.devplmt.XYZ.com)))";

  public static void main(String[] argv) throws Exception
  {
// Starting up the database:
    OracleDataSource ds = new OracleDataSource();
    Properties prop = new Properties();
    prop.setProperty("user","sys");
    prop.setProperty("password","manager");
    prop.setProperty("internal_logon","sysdba");
    prop.setProperty("prelim_auth","true");
    ds.setConnectionProperties(prop);
    ds.setURL(DB_URL);
    OracleConnection conn = (OracleConnection)ds.getConnection();
    conn.startup(OracleConnection.DatabaseStartupMode.NO_RESTRICTION);
    conn.close();
```

```
// Mounting and opening the database
    OracleDataSource ds1 = new OracleDataSource();
    Properties prop1 = new Properties();
    prop1.setProperty("user","sys");
    prop1.setProperty("password","manager");
    prop1.setProperty("internal_logon","sysdba");
    ds1.setConnectionProperties(prop1);
    ds1.setURL(DB_URL);
    OracleConnection conn1 = (OracleConnection)ds1.getConnection();
    Statement stmt = conn1.createStatement();
    stmt.executeUpdate("ALTER DATABASE MOUNT");
    stmt.executeUpdate("ALTER DATABASE OPEN");
    stmt.close();
    conn1.close();

// Shutting down the database
    OracleDataSource ds2 = new OracleDataSource();
    Properties prop = new Properties();
    prop.setProperty("user","sys");
    prop.setProperty("password","manager");
    prop.setProperty("internal_logon","sysdba");
    ds2.setConnectionProperties(prop);
    ds2.setURL(DB_URL);
    OracleConnection conn2 = (OracleConnection)ds2.getConnection();
    conn2.shutdown(OracleConnection.DatabaseShutdownMode.IMMEDIATE);
    Statement stmt1 = conn2.createStatement();
    stmt1.executeUpdate("ALTER DATABASE CLOSE NORMAL");
    stmt1.executeUpdate("ALTER DATABASE DISMOUNT");
    stmt1.close();
    conn2.shutdown(OracleConnection.DatabaseShutdownMode.FINAL);
    conn2.close();
  }
}
```

32
Diagnosability in JDBC

The diagnosabilty features of Oracle Database 12*c* Release 1 (12.1) enable you to diagnose problems in the applications that use Oracle JDBC drivers and the problems in the drivers themselves. They also reduce the effort required to develop and maintain Java applications that access an Oracle Database instance using Oracle JDBC drivers.

Oracle JDBC drivers provide the following diagnosabilty features that enable users to identify and fix problems in their JDBC applications:

- Logging
- Diagnosability Management

Note: The diagnosability features of the JDBC drivers are based on the standard `java.util.logging` framework and the `javax.management` MBean framework. Information about these standard frameworks is not covered in this document. For more information about these standard frameworks refer to

`http://www.oracle.com/technetwork/java/index.html`

Logging

This feature logs information about events that occur when JDBC driver code runs. Events can include user-visible events, such as SQL exceptions, running of SQL statements, and detailed JDBC internal events, such as entry to and exit from internal JDBC methods. Users can enable this feature to log specific events or all the events.

Prior to Oracle Database 11*g*, JDBC drivers supported J2SE 2.0 and 3.0. These versions of J2SE did not include `java.util.logging`. Therefore, the logging feature provided by JDBC driver releases prior to Oracle Database 11*g*, differs from the `java.util.logging` framework.

Starting from Oracle Database 11*g*, the JDBC drivers no longer support J2SE 2.0 and 3.0. Therefore, the logging feature of JDBC drivers makes full use of the standard `java.util.logging` package. The enhanced logging system makes effective use of log levels to enable users to restrict log output to things of interest. It logs specific classes of information more consistently, making it easier for the user to understand the log file.

This feature does not introduce new APIs or configuration files. Only new parameters are added to the existing standard `java.util.logging` configuration file. These

parameters are identical in use to the existing parameters and are intrinsic to using `java.util.logging`.

> **Note:** Oracle does not guarantee the exact content of the generated logs. To a large extent the log content is dependent on the details of the implementation. The details of the implementation change with every release, and therefore, the exact content of the logs are likely to change from release to release.

Enabling and Using JDBC Logging

Before you can start debugging your Java application, you must enable and configure JDBC logging. This section covers the steps you must perform to enable and use JDBC logging. It describes the following:

- Configuring the CLASSPATH
- Enabling Logging
- Configuring Logging
- Using Loggers
- Example

Configuring the CLASSPATH

Oracle ships several JAR files for each version of the JDBC drivers. The optimized JAR files do not contain any logging code and, therefore, do not generate any log output when used. To get log output, you must use the debug JAR files, which are indicated with a "_g" in the file name, like `ojdbc6_g.jar` or `ojdbc7_g.jar`. The debug JAR file must be included in the `CLASSPATH`.

> **Note:** Ensure that the debug JAR file, say `ojdbc6_g.jar` or `ojdbc7_g.jar`, is the only Oracle JDBC JAR file in the `CLASSPATH`.

Enabling Logging

You can enable logging in the following ways:

- Setting a Java system property

  You can enable logging by setting the `oracle.jdbc.Trace` system property.

  ```
  java -Doracle.jdbc.Trace=true ...
  ```

  Setting the system property enables global logging, which means that logging is enabled for the entire application. You can use global logging if you want to debug the entire application, or if you cannot or do not want to change the source code of the application.

- Programmatically

  You can programmatically enable or disable logging in the following way:

  First, get the `ObjectName` of the Diagnosability MBean. The `ObjectName` is

  ```
  com.oracle.jdbc:type=diagnosability,name=<loader>
  ```

  Here, `loader` is a unique name based on the class loader instance that loaded the Oracle JDBC drivers.

Note: The drivers can be loaded multiple times in a single VM. So, there can be multiple MBeans, each with a unique name.

Now, write the following lines of code:

```
ClassLoader l = oracle.jdbc.OracleDriver.getClassLoader();
String loader = l.getName() + "@" + l.hashCode();
// compute the ObjectName
javax.management.ObjectName name = new
javax.management.ObjectName("com.oracle.jdbc:type=diagnosability,
name="+loader);

// get the MBean server
javax.management.MBeanServer mbs =
java.lang.management.ManagementFactory.getPlatformMBeanServer();

// find out if logging is enabled or not
System.out.println("LoggingEnabled = " + mbs.getAttribute(name,
"LoggingEnabled"));

// enable logging
mbs.setAttribute(name, new javax.management.Attribute("LoggingEnabled", true));

// disable logging
mbs.setAttribute(name, new javax.management.Attribute("LoggingEnabled",
false));
```

Note:

- If the same class loader loads the JDBC drivers multiple times, then each subsequent MBean increments the value of the `l.hashCode()` method, so as to create a unique name. It may be problematic to identify which MBean is associated with which JDBC driver instance.
- If there is only one instance of the JDBC drivers loaded, then set the name to "*".

Programmatic enabling and disabling of logging helps you to control what parts of your application need to generate log output.

Note: Enabling logging using either of the methods would only generate a minimal log of serious errors. Usually this is not of much use. To generate a more useful and detailed log, you must configure `java.util.logging`.

Configuring Logging

To generate a useful and detailed log, you must configure `java.util.logging`. This can be done either through a configuration file or programmatically.

A sample configuration file, `OracleLog.properties`, is provided as part of the JDBC installation in the `demo` directory. It contains basic information about how to configure `java.util.logging` and provides some initial settings that you can start with. You

may use this sample file as is, edit the file and use it, rename the file and edit it, or create an entirely new file of any name.

To use a configuration file, you must identify it to the Java run-time. This can be done by setting a system property. For example:

```
java -Djava.util.logging.config.file=/jdbc/demo/OracleLog.properties.
```

It is read by the `java.util.logging` system. This file can reside anywhere.

You can use both `java.util.logging.config.file` and `oracle.jdbc.Trace` at the same time.

```
java -Djava.util.logging.config.file=/jdbc/demo/OracleLog.properties
-Doracle.jdbc.Trace=true
```

You can use the default `OracleLog.properties` file. It may or may not get you the desired output. You can also create and use your own configuration file by following these steps:

1. Create a file named `myConfig.properties`. You can use any name you choose.
2. Insert the following lines of text in the file:

   ```
   .level=SEVERE
   oracle.jdbc.level=INFO
   oracle.jdbc.handlers=java.util.logging.ConsoleHandler
   java.util.logging.ConsoleHandler.level=INFO
   java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
   ```

3. Save the file.
4. Set the system property to use this configuration file.

   ```
   java -Djava.util.logging.config.file=<filepath>/myConfig.properties ...
   ```

   filepath is the path of the folder where you have saved the `myConfig.properties` file.

You can also configure `java.util.logging` to dump the log output into a file. To do so, modify the configuration file as follows:

```
.level=SEVERE
oracle.jdbc.level=INFO
oracle.jdbc.handlers=java.util.logging.FileHandler
java.util.logging.FileHandler.level=INFO
java.util.logging.FileHandler.pattern=jdbc.log
java.util.logging.FileHandler.count=1
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
```

This will generate exactly the same log output and save it in a file named `jdbc.log` in the current directory.

You can control the amount of detail by changing the level settings. The defined levels from the least detailed to the most detailed are the following:

- `OFF`

  Turns off logging.

- `SEVERE`

  Logs SQLExceptions and internal errors.

- `WARNING`

Logs SQLWarnings and bad but not fatal internal conditions.

- `INFO`

  Logs infrequent but significant events and errors. It produces a relatively low volume of log messages.

- `CONFIG`

  Logs SQL strings that are executed.

- `FINE`

  Logs the entry and exit to every public method providing a detailed trace of JDBC operations. It produces a fairly high volume of log messages.

- `FINER`

  Logs calls to internal methods.

- `FINEST`

  Logs calls to high volume internal methods.

- `ALL`

  Logs all the details. This is the most detailed level of logging.

> **Note:** Levels more detailed than `FINE` generate huge log volumes.

In the example provided earlier, to reduce the amount of detail, change the `java.util.logging.FileHandler.level` setting from `ALL` to `INFO`:

```
java.util.logging.FileHandler.level=INFO
```

Although you can, it is not necessary to change the level of the `oracle.jdbc` logger. Setting the `FileHandler` level will control what log messages are dumped into the log file.

Using Loggers

Setting the level reduces all the logging output from JDBC. However, sometimes you need a lot of output from one part of the code and very little from other parts. To do that you must understand more about loggers.

Loggers exist in a tree structure defined by their names. The root logger is named "", the empty string. If you look at the first line of the configuration file you see `.level=SEVERE`. This is setting the level of the root logger. The next line is `oracle.jdbc.level=INFO`. This sets the level of the logger named `oracle.jdbc`. The `oracle.jdbc` logger is a member of the logger tree. Its parent is named `oracle`. The parent of the `oracle` logger is the root logger (the empty string).

Logging messages are sent to a particular logger, for example, `oracle.jdbc`. If the message passes the level check at that level, then the message is passed to the handler at that level, if any, and to the parent logger. So a log message sent to `oracle.log` is compared against that logger's level, `INFO` if you are following along. If the level is the same or less (less detailed) then it is sent to the FileHandler and to the parent logger, 'oracle'. Again it is checked against the level. If as in this case, the level is not set then it uses the parent level, `SEVERE`. If the message level is the same or less it is passed to the handler, which there is not one, and sent to the parent. In this case the parent in the root logger.

All this tree structure did not help you reduce the amount of output. What will help is that the JDBC drivers use several subloggers. If you restrict the log messages to one of the subloggers you will get substantially less output. The loggers used by Oracle JDBC drivers include the following:

- `oracle.jdbc`
- `oracle.jdbc.pool`
- `oracle.jdbc.rowset`
- `oracle.jdbc.xa`
- `oracle.sql`

> **Note:** The loggers used by the drivers may vary from release to release.

Example

Suppose you want to trace what is happening in the `oracle.sql` component and also want to capture some basic information about the rest of the driver. This is a more complex use of logging. The following are the entries in the `config` file:

```
     #
     # set levels
     #
.level=SEVERE
oracle.level=INFO
oracle.jdbc.driver.level=INFO
oracle.jdbc.pool.level=OFF
oracle.jdbc.util.level=OFF
oracle.sql.level=INFO
     #
     # configure handlers
     #
oracle.handlers=java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level=INFO
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```

Let us consider what each line in the configuration file is doing.

```
.level=SEVERE
```

Sets the logging level of the root logger to `SEVERE`. We do not want to see any logging from other, non-Oracle components unless something fails badly. Therefore, we set the default level for all loggers to `SEVERE`. Each logger inherits its level from its parent unless set explicitly. By setting the level of the root logger to `SEVERE` we ensure that all other loggers inherit that level except for the ones we set otherwise.

```
oracle.level=INFO
```

We want log output from both the `oracle.sql` and `oracle.jdbc.driver` loggers. Their common ancestor is `oracle`. Therefore, we set the level of the `oracle` logger to `INFO`. We will control the detail more explicitly at lower levels.

```
oracle.jdbc.driver.level=INFO
```

We only want to see the SQL execution from `oracle.jdbc.driver`. Therefore, we set the level to `INFO`. This is a fairly low volume level, but will help us to keep track of what our test is doing.

```
oracle.jdbc.pool.level=OFF
```

We are using a `DataSource` in our test and do not want to see all of that logging. Therefore, we turn it `OFF`.

```
oracle.jdbc.util.level=OFF
```

We do not want to see the logging from the `oracle.jdbc.util` package. If we were using XA or row sets we would turn them off as well.

```
oracle.sql.level=INFO
```

We want to see what is happening in `oracle.sql`. Therefore, we set the level to `INFO`. This provides a lot of information about the public method calls without overwhelming detail.

```
oracle.handlers=java.util.logging.ConsoleHandler
```

We are going to dump everything to `stderr`. When we run the test we will redirect `stderr` to a file.

```
java.util.logging.ConsoleHandler.level=INFO
```

We want to dump everything to the console which is `System.err`. In this case, we are doing the filtering with the loggers rather than the handler.

```
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```

We will use a simple, more or less human readable format.

When you run your test with this configuration file, you will get moderately detailed information from the `oracle.sql` package, a little bit of information from the core driver code, and nothing from any other code.

You can also use `XMLFormatter` for sending logs to Oracle Support.

You can implement and use a custom `java.util.logging.Filter` to obtain finer control of the data captured in the logs. This is a standard `java.util.logging` feature and is documented in the JSE JavaDoc. A custom Filter enables you to:

- Capture only one thread in multithreaded applications
- Capture intermittent errors in long running applications

Performance, Scalability, and Security Issues

Although the logging feature enables you to trace or debug your application and generate detail log output, it has certain performance, scalability, and security issues.

Performance and Scalability Issues

Logging has substantial impact on performance. However, JDBC logging is generally not enabled in production systems. When logging is disabled, it will have no impact on performance.

It also has a negative impact on scalability. Logging involves protected access to a number of shared resources resulting in severely reduced scalability. This is intrinsic to the `java.util.logging` framework. However, in a typical production system, JDBC logging is not enabled and, therefore, will not have an impact on scalability.

Security Concerns

When full logging is enabled, it is almost guaranteed that all sensitive information will be exposed in the log files. This is intrinsic to the logging feature. However, only certain JDBC JAR files include the JDBC logging feature. The following JAR files include full logging and should not be used in a sensitive environment:

- `ojdbc6_g.jar`
- `ojdbc6dms_g.jar`
- `ojdbc7_g.jar`
- `ojdbc7dms_g.jar`

The following JAR files include a limited logging capability:

- `ojdbc6dms.jar`
- `ojdbc7dms.jar`

> **Note:** Database user names and passwords do not appear in log files created by these JAR files. However, sensitive user data that is part of a SQL statement, a defined value, or a bind value can appear in a log created using one of these JAR files.

Diagnosability Management

The JDBC diagnosability management feature introduces an MBean, `oracle.jdbc.driver.OracleDiagnosabilityMBean`. This MBean provides means to enable and disable JDBC logging.

> **See Also:** For information about the `OracleDiagnosabilityMBean` API, refer to the JDBC Javadoc.

In future releases, the MBean will be enhanced to provide additional statistics about JDBC internals.

Security Concerns

This feature can enable JDBC logging. Enabling JDBC logging does not require any special permission. However, once logging is enabled, generating any log output requires the standard Java permission `LoggingPermission`. Without this permission, any JDBC operation that would generate log output will throw a security exception. This is a standard Java mechanism.

33

JDBC DMS Metrics

DMS metrics are used to measure the performance of application components.

This chapter discusses the following topics:

- Overview of JDBC DMS Metrics
- Determining the Type of Metric to Be Generated
- Generating the SQLText Metric
- Accessing DMS Metrics Using JMX

Note: There is another kind of metrics called end-to-end metrics. End-to-end metrics are used for tagging application activity from the entry into the application code through JDBC to the database and back.

JDBC supports the following end-to-end metrics:

- `Action`
- `ClientId`
- `ExecutionContextId`
- `Module`
- `State`

For earlier releases, to work with the preceding metrics, you could use the `setEndToEndMetrics` and `getEndToEndMetrics` methods of the `oracle.jdbc.OracleConnection` interface. However, starting from Oracle Database 12*c* Release 1 (12.1), these methods have been deprecated. Oracle recommends to use the `setClientInfo` and `getClientInfo` methods instead of the `setEndToEndMetrics` and `getEndToEndMetrics` methods.

In Oracle Database 10*g*, Oracle Java Database Connectivity (JDBC) supports end-to-end metrics. In Oracle Database 12*c* Release 1 (12.1), an application can set the end-to-end metrics directly only when it does not use a DMS-enabled JAR files. But, if the application uses a DMS-enabled JAR file, the end-to-end metrics can be set only through DMS.

For more information about end-to-end metrics, refer to *Oracle Database JDBC Java API Reference.*

Caution: Oracle strongly recommends using DMS metrics, if the application uses a DMS-enabled JAR file.

Overview of JDBC DMS Metrics

DMS metrics enable application and system developers to measure and export customized performance metrics for specific software components. All DMS metrics are available in the following DMS-enabled JAR files:

- `ojdbc6dms.jar`
- `ojdbc6dms_g.jar`
- `ojdbc7dms.jar`
- `ojdbc7dms_g.jar`

Any other JDBC JAR files do not generate any DMS metrics. The metrics generated in Oracle Database 12*c* Release 1 (12.1) are different from 10.2, 10.1, 9.2, and earlier versions of Oracle JDBC as it makes no attempt to retain compatibility with earlier versions. There are also no compatibility modes. A system that is dependent on the exact details of the DMS metrics generated by earlier versions of JDBC may have unexpected behavior when processing the metrics generated by Oracle JDBC 12*c*. This is by design and cannot be changed.

Statement metrics can be reported consolidated for all statements in a connection or individually for each statement. All DMS metrics, except those related to individual statements, are enabled at all times.

Note: You can enable or disable the `SQLText` statement metric. It is disabled by default. If enabled, it is enabled for all statements.

Determining the Type of Metric to Be Generated

To determine whether to use consolidated or individual metrics, JDBC checks the DMSConsole sensor weight. If the sensor weight is less than or equal to `DMSConsole.NORMAL`, then JDBC generates consolidated statement metrics. If the sensor weight is greater than `DMSConsole.NORMAL`, then JDBC generates individual statement metrics.

JDBC checks the DMSConsole sensor weight when creating a Prepared or Callable statement and depending on the sensor weight at the time the statement is created, the metrics are generated. Changing the value of the sensor weight, after the statement has been created, does not cause a statement to switch between consolidated and individual metrics.

Note: In the presence of Statement caching, it may appear that changing sensor weight has no impact as statements are retrieved from the cache rather than created anew.

There is only one list of statement metrics that is generated for both consolidated and individual statement metrics. The only difference between these two lists is the aggregation of the statements. When individual statement metrics are generated, one set of metrics is generated for each distinct statement object created by the JDBC

driver. On the other hand, when consolidated statement metrics are generated, all statements created by a given connection use the same set of statement metrics.

For example, consider an 'execute' phase event. If individual statement metrics are used, then each statement created will have a distinct 'execute' phase event. So, from two such statements, it will be possible to distinguish the execution statistics for the two separate statements. If one has an execution time of 1 second and the other an execution time of 3 seconds, then there will be two distinct 'execute' phase events, one with a total time and average of 1 second and the other with a total time and average of 3 seconds. But, if consolidated statement metrics are used, all statements will use the single 'execute' phase event common to the connection. So, from two such statements created by the same connection, it will not be possible to distinguish the execution statistics for the two statements. If one has an execution time of 1 second and the other an execution time of 3 seconds, then the common 'execute' phase event will report a total execution time of 4 seconds and an average of 2 seconds.

Generating the SQLText Metric

Depending on the version of DMS, there are two mechanisms for determining the generating of the SQLText statement metrics:

- If the 12*c* version of the DMS JAR file is present in the classpath environment variable, then JDBC checks the DMS update SQL text flag. If this flag is true, then the SQLText metric is updated.
- If the 12*c* version of the DMS JAR file is not present in the classpath environment variable, then JDBC uses the value of the DMSStatementMetrics connection property. If this statement property is true, then SQLText metric is updated. The default value of this connection property is false.

Whether or not the SQLText metric will be generated is independent of the use of the type of statement metrics used, that is, individual statement metrics or consolidated statement metrics.

Accessing DMS Metrics Using JMX

JMX (Java Management Extensions) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices, service-oriented networks, and the JVM (Java Virtual Machine). You can easily access DMS metrics at run time using a management application that supports JMX. For more information about using JMX to access DMS data, go to the following URL
http://www.oracle.com/technetwork/middleware/toplink/overview/index.html

> **See Also:** *Oracle Database Java Developer's Guide* for more information about JMX

Part IX
Appendixes

This part consists of appendixes that discuss Java Database Connectivity (JDBC) reference information, tips for coding JDBC applications, JDBC error messages, and troubleshooting JDBC applications.

Part IX contains the following appendixes:

A

JDBC Reference Information

This appendix contains detailed Java Database Connectivity (JDBC) reference information, including the following topics:

- Supported SQL-JDBC Data Type Mappings
- Supported SQL and PL/SQL Data Types
- Using PL/SQL Types
- Using Embedded JDBC Escape Syntax
- Oracle JDBC Notes and Limitations

Supported SQL-JDBC Data Type Mappings

Table 11–1 describes the default mappings between Java classes and SQL data types supported by Oracle JDBC drivers. Compare the contents of the JDBC Type Codes, Standard Java Types, and SQL Data Types columns in Table 11–1 with the contents of Table A–1.

Table A–1 lists all the possible Java types to which a given SQL data type can be validly mapped. Oracle JDBC drivers will support these nondefault mappings. For example, to materialize SQL `CHAR` data in an `oracle.sql.CHAR` object, use the `getCHAR` method. To materialize it as a `java.math.BigDecimal` object, use the `getBigDecimal` method.

Note: For classes where `oracle.jdbc.OracleData` appears in italic, these can be generated by JPublisher.

Table A–1 Valid SQL Data Type-Java Class Mappings

SQL data type	Java types
`CHAR`, `VARCHAR2`, `LONG`	`java.lang.String`
	`oracle.sql.CHAR`

Table A–1 (Cont.) Valid SQL Data Type-Java Class Mappings

SQL data type	Java types
NUMBER	boolean
	char
	byte
	short
	int
	long
	float
	double
	java.lang.Byte
	java.lang.Short
	java.lang.Integer
	java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
	oracle.sql.NUMBER
BINARY_INTEGER	boolean
	char
	byte
	short
	int
	long
BINARY_FLOAT	oracle.sql.BINARY_FLOAT
BINARY_DOUBLE	oracle.sql.BINARY_DOUBLE
DATE	oracle.sql.DATE
RAW	oracle.sql.RAW
BLOB	oracle.jdbc.OracleBlob[1]
CLOB	oracle.jdbc.OracleClob[2]
BFILE	oracle.sql.BFILE
ROWID	oracle.sql.ROWID
TIMESTAMP	oracle.sql.TIMESTAMP
TIMESTAMP WITH TIME ZONE	oracle.sql.TIMESTAMPTZ
TIMESTAMP WITH LOCAL TIME ZONE	oracle.sql.TIMESTAMPLTZ
ref cursor	java.sql.ResultSet
	sqlj.runtime.ResultSetIterator
user defined named types, ADTs	oracle.jdbc.OracleStruct[3]
opaque named types	oracle.jdbc.OracleOpaque[4]

Table A–1 (Cont.) Valid SQL Data Type-Java Class Mappings

SQL data type	Java types
nested tables and VARRAY named types	`oracle.jdbc.OracleArray`[5]
references to named types	`oracle.jdbc.OracleRef`[6]

[1] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.BLOB` class is deprecated and replaced with the `oracle.jdbc.OracleBlob` interface.

[2] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.CLOB` class is deprecated and replaced with the `oracle.jdbc.OracleClob` interface.

[3] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.STRUCT` class is deprecated and replaced with the `oracle.jdbc.OracleStruct` interface.

[4] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.OPAQUE` class is deprecated and replaced with the `oracle.jdbc.OracleOpaque` interface.

[5] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.ARRAY` class is deprecated and replaced with the `oracle.jdbc.OracleArray` interface.

[6] Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.REF` class is deprecated and replaced with the `oracle.jdbc.OracleRef` interface.

Note:

- The type `UROWID` is not supported.
- The `oracle.sql.Datum` class is abstract. The value passed to a parameter of type `oracle.sql.Datum` must be of the Java type corresponding to the underlying SQL type. Likewise, the value returned by a method with return type `oracle.sql.Datum` must be of the Java type corresponding to the underlying SQL type.

Supported SQL and PL/SQL Data Types

The tables in this section list SQL and PL/SQL data types, and whether Oracle JDBC drivers support them. Table A–2 describes Oracle JDBC driver support for SQL data types.

Table A–2 Support for SQL Data Types

SQL Data Type	Supported by JDBC Drivers?
BFILE	yes
BLOB	yes
CHAR	yes
CLOB	yes
DATE	yes
NCHAR	no[1]
NCHAR VARYING	no
NUMBER	yes
NVARCHAR2	yes[2]
RAW	yes
REF	yes
ROWID	yes

Table A–2 (Cont.) Support for SQL Data Types

SQL Data Type	Supported by JDBC Drivers?
UROWID	no
VARCHAR2	yes

[1] The NCHAR type is supported indirectly. There is no corresponding java.sql.Types type, but if your application calls the formOfUse(NCHAR) method, then this type can be accessed.

[2] In JSE 6, the NVARCHAR2 type is supported directly. In J2SE 5.0, the NVARCHAR2 type is supported indirectly. There is no corresponding java.sql.Types type, but if your application calls the formOfUse(NCHAR) method, then this type can be accessed.

Table A–3 describes Oracle JDBC support for the ANSI-supported SQL data types.

Table A–3 Support for ANSI-92 SQL Data Types

ANSI-Supported SQL Data Type	Supported by JDBC Drivers?
CHARACTER	yes
DEC	yes
DECIMAL	yes
DOUBLE PRECISION	yes
FLOAT	yes
INT	yes
INTEGER	yes
NATIONAL CHARACTER	no
NATIONAL CHARACTER VARYING	no
NATIONAL CHAR	yes
NATIONAL CHAR VARYING	no
NCHAR	yes
NCHAR VARYING	no
NUMERIC	yes
REAL	yes
SMALLINT	yes
VARCHAR	yes

Table A–4 describes Oracle JDBC driver support for SQL User-Defined types.

Table A–4 Support for SQL User-Defined Types

SQL User-Defined type	Supported by JDBC Drivers?
OPAQUE	yes
Reference types	yes
Object types (`JAVA_OBJECT`)	yes
Nested table types and VARRAY types	yes

Table A–5 describes Oracle JDBC driver support for PL/SQL data types. Note that PL/SQL data types include these categories:

- Scalar types
- Scalar character types, which includes `DATE` data type
- Composite types
- Reference types
- Large object (LOB) types

Table A–5 Support for PL/SQL Data Types

PL/SQL Data Type	Supported by JDBC Drivers?
Scalar Types:	
BINARY INTEGER	yes
DEC	yes
DECIMAL	yes
DOUBLE PRECISION	yes
FLOAT	yes
INT	yes
INTEGER	yes
NATURAL	yes
NATURAL*n*	no
NUMBER	yes
NUMERIC	yes
PLS_INTEGER	yes
POSITIVE	yes
POSITIVE*n*	no
REAL	yes
SIGNTYPE	yes
SMALLINT	yes
Scalar Character Types:	
CHAR	yes
CHARACTER	yes
LONG	yes
LONG RAW	yes
NCHAR	no (see Note)
NVARCHAR2	no (see Note)
RAW	yes
ROWID	yes
STRING	yes
UROWID	no
VARCHAR	yes

Table A–5 (Cont.) Support for PL/SQL Data Types

PL/SQL Data Type	Supported by JDBC Drivers?
VARCHAR2	yes
DATE	yes
Composite Types:	
RECORD	no
TABLE	no
VARRAY	yes
Reference Types:	
REF CURSOR types	yes
object reference types	yes
LOB Types:	
BFILE	yes
BLOB	yes
CLOB	yes
NCLOB	yes

Note:

- The types `NATURAL`, `NATURAL`*n*, `POSITIVE`, `POSITIVE`*n*, and `SIGNTYPE` are subtypes of `BINARY INTEGER`.
- The types `DEC`, `DECIMAL`, `DOUBLE PRECISION`, `FLOAT`, `INT`, `INTEGER`, `NUMERIC`, `REAL`, and `SMALLINT` are subtypes of `NUMBER`.
- The types `NCHAR` and `NVARCHAR2` are supported indirectly. There is no corresponding `java.sql.Types` type, but if your application calls `formOfUse(NCHAR)`, then these types can be accessed. Refer to "NCHAR, NVARCHAR2, NCLOB and the defaultNChar Property" on page 19-3 for details.

Using PL/SQL Types

Starting from Oracle Database 12*c* Release 1 (12.1), you can map schema-level PL/SQL types as generic `java.sql.Struct` type and PL/SQL collections as `java.sql.Array` types. So, instead of creating schema-level types that are mapped to PL/SQL package types for binding, you can describe and bind PL/SQL types using only the JDBC APIs.

For example, you can call the `Connection.createStruct(type_name)` method to first create a descriptor that can be used to describe a PL/SQL type and then to create a new `STRUCT` representation of this type on the client. In Oracle Database 12*c* Release 1 (12.1), you can reuse this API by specifying `type_name` as `"schema.package.typename"` or `"package.typename"`.

All PL/SQL package types are mapped to a system-wide unique name that can be used by JDBC to retrieve the server-side type metadata. The name is in the following form:

```
[SCHEMA.]<PACKAGE>.<TYPE>
```

Note: If the schema is the same as the package name, and if there is a type with the same name as the PL/SQL type, then it will not be able to identify an object with the two part name format, that is, <package>.<type>. In such cases, you must use three part names <schema>.<package>.<type>.

Example A–1 explains how to bind types declared in PL/SQL packages.

Example A–1 Binding Types Declared In PL/SQL Packages

```
/*
---------------------------
# Perform the following SQL operations prior to running this sample
---------------------------
conn HR/hr;
create or replace package TEST_PKG is
   type V_TYP is varray(10) of varchar2(200);
   type R_TYP is record(c1 pls_integer, c2 varchar2(100));
   procedure VARR_PROC(p1 in V_TYP, p2 OUT V_TYP);
   procedure REC_PROC(p1 in R_TYP, p2 OUT R_TYP);
end;
/
create or replace package body TEST_PKG is
 procedure VARR_PROC(p1 in V_TYP, p2 OUT V_TYP) is
  begin
    p2 := p1;
  end;
  procedure REC_PROC(p1 in R_TYP, p2 OUT R_TYP) is
  begin
    p2 := p1;
  end;
end;
/
*/

import java.sql.Array;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Struct;
import java.sql.Types;

import oracle.jdbc.OracleConnection;
public class PLSQLTypesSample
{
  public static void main(String[] args) throws SQLException
  {
    System.out.println("begin...");
    Connection conn = null;
    oracle.jdbc.pool.OracleDataSource ods = new
oracle.jdbc.pool.OracleDataSource();
    ods.setURL("jdbc:oracle:oci:localhost:5521:orcl");
    ods.setUser("HR");
    ods.setPassword("hr");
    //get connection
    conn = ods.getConnection();

    //call procedure TEST_PKG.VARR_PROC
```

```
    CallableStatement cstmt = null;
    try {
      cstmt = conn.prepareCall("{ call TEST_PKG.VARR_PROC(?,?) }");
      //PLSQL VARRAY type binding
      Array arr = ((OracleConnection)conn).createArray("TEST_PKG.V_TYP", new
String[]{"A", "B"});
      cstmt.setArray(1, arr);
      cstmt.registerOutParameter(2, Types.ARRAY, "TEST_PKG.V_TYP");
      cstmt.execute();
      //get PLSQL VARRAY type out parameter value
      Array outArr = cstmt.getArray(2);
      //...
    }
    catch( Exception e) {
      e.printStackTrace();
    }finally {
      if (cstmt != null)
        cstmt.close();
    }

    //call procedure TEST_PKG.REC_PROC
    try {
      cstmt = conn.prepareCall("{ call TEST_PKG.REC_PROC(?,?) }");
      //PLSQL RECORD type binding
      Struct struct = conn.createStruct("TEST_PKG.R_TYP", new Object[]{12345,
"B"});
      cstmt.setObject(1, struct);
      cstmt.registerOutParameter(2, Types.STRUCT, "TEST_PKG.R_TYP");
      cstmt.execute();
      //get PLSQL RECORD type out parameter value
      Struct outStruct = (Struct)cstmt.getObject(2);
      //...
    }
    catch( Exception e) {
      e.printStackTrace();
    }finally {
      if (cstmt != null)
        cstmt.close();
    }

    if (conn != null)
      conn.close();

    System.out.println("done!");
  }
}
```

Creating Java level objects for each row using %ROWTYPE Attribute

You can create Java-level objects using the `%ROWTYPE` attribute. In this case, each row of the table is created as a `java.sql.Struct` object. For example, if you have a package `pack1` with the following specification:

> **See Also:** *Oracle Database PL/SQL Language Reference* for more information about the `%ROWTYPE` attribute

```
CREATE OR REPLACE PACKAGE PACK1 AS
  TYPE EMPLOYEE_ROWTYPE_ARRAY IS TABLE OF EMPLOYEES%ROWTYPE;
END PACK1;
```

```
/
```

The following code snippet shows how you can retrieve the value of the EMPLOYEE_ROWTYPE_ARRAY array using JDBC APIs:

Example A–2 Creating Struct Objects for Database Table Rows

```
CallableStatement cstmt = conn.prepareCall("BEGIN SELECT * BULK COLLECT INTO :1
FROM EMPLOYEE; END;");
cstmt.registerOutParameter(1,OracleTypes.ARRAY, "PACK1.EMPLOYEE_ROWTYPE_ARRAY");
cstmt.execute();
Array a = cstmt.getArray(1);
```

Example A–2 returns a java.sql.Array of java.sql.Struct objects, where every Struct element represents one row of the EMPLOYEES table.

Using Embedded JDBC Escape Syntax

Oracle JDBC drivers support some embedded JDBC escape syntax, which is the syntax that you specify between curly braces. The current support is basic.

> **Note:** JDBC escape syntax was previously known as SQL92 Syntax or SQL92 escape syntax.

This section describes the support offered by the drivers for the following constructs:

- Time and Date Literals
- Scalar Functions
- LIKE Escape Characters
- MATCH_RECOGNIZE Clause
- Outer Joins
- Function Call Syntax

Where driver support is limited, these sections also describe possible workarounds.

Disabling Escape Processing

The processing for JDBC escape syntax is enabled by default, which results in the JDBC driver performing escape substitution before sending the SQL code to the database. If you want the driver to use regular Oracle SQL syntax, which is more efficient than JDBC escape syntax processing, then use this statement:

```
stmt.setEscapeProcessing(false);
```

Time and Date Literals

Databases differ in the syntax they use for date, time, and timestamp literals. JDBC supports dates and times written only in a specific format. This section describes the formats you must use for date, time, and timestamp literals in SQL statements.

Date Literals

The JDBC drivers support date literals in SQL statements written in the format:

```
{d 'yyyy-mm-dd'}
```

Where `yyyy-mm-dd` represents the year, month, and day. For example:

```
{d '1995-10-22'}
```

The JDBC drivers will replace this escape clause with the equivalent Oracle representation: "22 OCT 1995".

The following code snippet contains an example of using a date literal in a SQL statement.

```
// Connect to the database
// You can put a database name after the @ sign in the connection URL.
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci:@");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();

// Create a Statement
Statement stmt = conn.createStatement ();

// Select the first name column from the employees table where the hire date is
Jan-23-1982
ResultSet rset = stmt.executeQuery
                 ("SELECT first_name FROM employees WHERE hire_date = {d
'1982-01-23'}");

// Iterate through the result and print the employee names
while (rset.next ())
   System.out.println (rset.getString (1));
```

Time Literals

The JDBC drivers support time literals in SQL statements written in the format:

```
{t 'hh:mm:ss'}
```

where, `hh:mm:ss` represents the hours, minutes, and seconds. For example:

```
{t '05:10:45'}
```

The JDBC drivers will replace this escape clause with the equivalent Oracle representation: "05:10:45".

If the time is specified as:

```
{t '14:20:50'}
```

Then the equivalent Oracle representation would be "14:20:50", assuming the server is using a 24-hour clock.

This code snippet contains an example of using a time literal in a SQL statement.

```
ResultSet rset = stmt.executeQuery
                 ("SELECT first_name FROM employees WHERE hire_date = {t
'12:00:00'}");
```

Timestamp Literals

The JDBC drivers support timestamp literals in SQL statements written in the format:

```
{ts 'yyyy-mm-dd hh:mm:ss.f...'}
```

where `yyyy-mm-dd hh:mm:ss.f...` represents the year, month, day, hours, minutes, and seconds. The fractional seconds portion (`.f...`) is optional and can be omitted. For example: `{ts '1997-11-01 13:22:45'}` represents, in Oracle format, NOV 01 1997 13:22:45.

This code snippet contains an example of using a timestamp literal in a SQL statement.

```
ResultSet rset = stmt.executeQuery
    ("SELECT first_name FROM employees WHERE hire_date = {ts '1982-01-23
12:00:00'}");
```

Mapping SQL DATE Data type to Java

Oracle Database 8*i* and earlier versions did not support `TIMESTAMP` data, but Oracle `DATE` data used to have a time component as an extension to the SQL standard. So, Oracle Database 8*i* and earlier versions of JDBC drivers mapped `oracle.sql.DATE` to `java.sql.Timestamp` to preserve the time component. Starting with Oracle Database 9.0.1, `TIMESTAMP` support was included and 9*i* JDBC drivers started mapping `oracle.sql.DATE` to `java.sql.Date`. This mapping was incorrect as it truncated the time component of Oracle `DATE` data. To overcome this problem, Oracle Database 11*g* Release 1 introduced a new flag `mapDateToTimestamp`. The default value of this flag is `true`, which means that by default the drivers will correctly map `oracle.sql.DATE` to `java.sql.Timestamp`, retaining the time information. If you still want the incorrect but 10*g* compatible `oracle.sql.DATE` to `java.sql.Date` mapping, then you can get it by setting the value of `mapDateToTimestamp` flag to `false`.

Note:

- Since Oracle Database 11*g*, if you have an index on a DATE column to be used by a SQL query, then to obtain faster and accurate results, you must use the `setObject` method in the following way:

  ```
  Date d = parseIsoDate(val);
  Timestamp t = new Timestamp(d.getTime());
  stmt.setObject(pos, new oracle.sql.DATE(t, (Calendar)UTC_
  CAL.clone()));
  ```

  This is because if you use the `setDate` method, then the time component of the Oracle DATE data will be lost and if you use the `setTimestamp` method, then the index on the DATE column will not be used.

- To overcome the problem of `oracle.sql.DATE` to `java.sql.Date` mapping, Oracle Database 9.2 introduced a flag, `V8Compatible`. The default value of this flag was `false`, which allowed the mapping of Oracle `DATE` data to `java.sql.Date` data. But, users could retain the time component of the Oracle `DATE` data by setting the value of this flag to `true`. This flag is desupported since 11*g* because it controlled Oracle Database 8*i* compatibility, which is no longer supported.

Scalar Functions

Oracle JDBC drivers do not support all scalar functions. To find out which functions the drivers support, use the following methods supported by the Oracle-specific

`oracle.jdbc.OracleDatabaseMetaData` class and the standard Java `java.sql.DatabaseMetadata` interface:

- `getNumericFunctions()`

  Returns a comma-delimited list of math functions supported by the driver. For example, `ABS`, `COS`, `SQRT`.

- `getStringFunctions()`

  Returns a comma-delimited list of string functions supported by the driver. For example, `ASCII`, `LOCATE`.

- `getSystemFunctions()`

  Returns a comma-delimited list of system functions supported by the driver. For example, `DATABASE`, `USER`.

- `getTimeDateFunctions()`

  Returns a comma-delimited list of time and date functions supported by the driver. For example, `CURDATE`, `DAYOFYEAR`, `HOUR`.

Note: Oracle JDBC drivers support `fn`, the function keyword.

LIKE Escape Characters

The characters `%` and _ have special meaning in SQL `LIKE` clauses. You use `%` to match zero or more characters and _ to match exactly one character. If you want to interpret these characters literally in strings, then you precede them with a special escape character. For example, if you want to use ampersand (`&`) as the escape character, then you identify it in the SQL statement as:

```
Statement stmt = conn.createStatement ();

// Select the empno column from the emp table where the ename starts with '_'
ResultSet rset = stmt.executeQuery
          ("SELECT empno FROM emp WHERE ename LIKE '&_%' {ESCAPE '&'}");

// Iterate through the result and print the employee numbers
while (rset.next ())
   System.out.println (rset.getString (1));
```

Note: If you want to use the backslash character (\) as an escape character, then you must enter it twice, that is, \\. For example:

```
ResultSet rset = stmt.executeQuery("SELECT empno FROM emp
                WHERE ename LIKE '\\_%' {escape '\\'}");
```

MATCH_RECOGNIZE Clause

The `?` character is used as a token in `MATCH_RECOGNIZE` clause in Oracle Database 11*g* and later versions. As the JDBC standard defines the `?` character as a parameter marker, the JDBC Driver and the Server SQL Engine cannot distinguish between different uses of the same token.

In earlier versions of JDBC Driver, if you want to interpret the `?` character as a `MATCH_RECOGNIZE` token and not as a parameter marker, then you must use a `Statement`

instead of a `PreparedStatement` and disable escape processing. However, starting from Oracle Database 12*c* Release 1 (12.1.0.2), you can use the '`{\ ... \}`' syntax while using the `?` character, so that the JDBC driver does not process it as a parameter marker and allows the SQL engine to process it. The following code snippet shows how to use the '`{\ ... \}`' syntax:

```
String sql =
  "select T.firstW, T.lastZ, ? " +  // use of parameter marker
  "from tkpattern_S11 " +
  "MATCH_RECOGNIZE ( " +
  "    MEASURES A.c1 as firstW, last(Z.c1) as lastZ " +
  "    ALL MATCHES " +
  "    PATTERN(A{\?\} X{\*?\} Y{\+?\} Z{\??\}) " +  // use of escape sequence
  "    DEFINE " +
  "        X as X.c2 > prev(X.c2), " +
  "        Y as Y.c2 < prev(Y.c2), " +
  "        Z as Z.c2 > prev(Z.c2)" +
  ") as T";
PreparedStatement ps = conn.prepareStatatement(sql);
ps.setString(1, "test");
ResultSet rs = ps.executeQuery();
```

See Also: "Disabling Escape Processing" on page A-9

Outer Joins

Oracle JDBC drivers do not support the outer join syntax. The workaround is to use Oracle outer join syntax:

Instead of:

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
     ("SELECT ename, dname
       FROM {OJ dept LEFT OUTER JOIN emp ON dept.deptno = emp.deptno}
       ORDER BY ename");
```

Use Oracle SQL syntax:

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
     ("SELECT ename, dname
       FROM emp b, dept a WHERE a.deptno = b.deptno(+)
       ORDER BY ename");
```

Function Call Syntax

Oracle JDBC drivers support the following procedure and function call syntax:

Procedure calls:

```
{ call procedure_name (argument1, argument2,...) }
```

Function calls:

```
{ ? = call procedure_name (argument1, argument2,...) }
```

JDBC Escape Syntax to Oracle SQL Syntax Example

You can write a simple program to translate JDBC escape syntax to Oracle SQL syntax. The following program prints the comparable Oracle SQL syntax for statements using JDBC escape syntax for function calls, date literals, time literals, and timestamp literals. In the program, the `oracle.jdbc.OracleSql` class `parse()` method performs the conversions.

```
public class Foo
{
   static oracle.jdbc.OracleDriver driver = new oracle.jdbc.OracleDriver();
   public static void main (String args[]) throws Exception
   {
      show ("{call foo(?, ?)}");
      show ("{? = call bar (?, ?)}");
      show ("{d '1998-10-22'}");
      show ("{t '16:22:34'}");
      show ("{ts '1998-10-22 16:22:34'}");
   }

   public static void show (String s) throws Exception
   {
      System.out.println (s + " => " +
         driver.processSqlEscapes(s));
   }
}
```

The following code is the output that prints the comparable SQL syntax.

```
{call foo(?, ?)} => BEGIN foo(:1, :2); END;
{? = call bar (?, ?)} => BEGIN :1 := bar (:2, :3); END;
{d '1998-10-22'} => TO_DATE ('1998-10-22', 'YYYY-MM-DD')
{t '16:22:34'} => TO_DATE ('16:22:34', 'HH24:MI:SS')
{ts '1998-10-22 16:22:34'} => TO_TIMESTAMP ('1998-10-22 16:22:34', 'YYYY-MM-DD
HH24:MI:SS.FF')
```

Oracle JDBC Notes and Limitations

The following limitations exist in the Oracle JDBC implementation, but all of them are either insignificant or have easy workarounds. This section covers the following topics:

- CursorName
- JDBC Outer Join Escapes
- IEEE 754 Floating Point Compliance
- Catalog Arguments to DatabaseMetaData Calls
- SQLWarning Class
- Executing DDL Statements
- Binding Named Parameters

CursorName

Oracle JDBC drivers do not support the `get getCursorName` and `setCursorName` methods, because there is no convenient way to map them to Oracle constructs. Oracle recommends using `ROWID` instead.

See Also: "Oracle ROWID Type" on page 4-13 for more information about how to use and manipulate `ROWID`s.

JDBC Outer Join Escapes

Oracle JDBC drivers do not support JDBC outer join escapes. Use Oracle SQL syntax with + instead.

See Also: "Using Embedded JDBC Escape Syntax" on page A-9

IEEE 754 Floating Point Compliance

The arithmetic for the Oracle `NUMBER` type does not comply with the IEEE 754 standard for floating-point arithmetic. Therefore, there can be small disagreements between the results of computations performed by Oracle and the same computations performed by Java.

Oracle stores numbers in a format compatible with decimal arithmetic and guarantees 38 decimal digits of precision. It represents zero, minus infinity, and plus infinity exactly. For each positive number it represents, it represents a negative number of the same absolute value.

It represents every positive number between 10^{-30} and $(1 - 10^{-38}) * 10^{126}$ to full 38-digit precision.

Catalog Arguments to DatabaseMetaData Calls

Certain `DatabaseMetaData` methods define a `catalog` parameter. This parameter is one of the selection criteria for the method. Oracle does not have multiple catalogs, but it does have packages.

See Also: "Reporting DatabaseMetaData TABLE_REMARKS" on page 21-21 for information about how Oracle JDBC drivers treat the `catalog` argument.

SQLWarning Class

The `java.sql.SQLWarning` class provides information about a database access warning. Warnings typically contain a description of the warning and a code that identifies the warning. Warnings are silently chained to the object whose method caused it to be reported. Oracle JDBC drivers generally do not support `SQLWarning`. As an exception to this, scrollable result set operations do generate SQL warnings, but the `SQLWarning` instance is created on the client, not in the database.

See Also: "Processing SQL Exceptions" on page 2-20

Executing DDL Statements

You must execute Data Definition Language (DDL) statements with `Statement` objects. If you use `PreparedStatements` objects or `CallableStatements` objects, then the DDL statement takes effect only on the first execution. This can cause unexpected behavior if the SQL statements are in a statement cache.

Binding Named Parameters

Binding by name is not supported when using the `setXXX` methods. Under certain circumstances, previous versions of Oracle JDBC drivers have allowed binding

statement variables by name when using the `set`*`XXX`* methods. In the following statement, the named variable `EmpId` would be bound to the integer `314159`.

```
PreparedStatement p = conn.prepareStatement
  ("SELECT name FROM emp WHERE id = :EmpId");
  p.setInt(1, 314159);
```

This capability to bind by name using the `set`*`XXX`* methods is not part of the JDBC specification, and Oracle does not support it. The JDBC drivers can throw a `SQLException` or produce unexpected results. Starting from Oracle Database 10*g* JDBC drivers, bind by name is supported using the `set`*`XXX`*`AtName` methods.

> **See Also:** "Interface oracle.jdbc.OracleCallableStatement" on page 4-22 and "Interface oracle.jdbc.OraclePreparedStatement" on page 4-21

The bound values are not copied by the drivers until you call the `execute` method. So, changing the bound value before calling the `execute` method could change the bound value. For example, consider the following code snippet:

```
PreparedStatement p;
.......
Date d = new Date(1181676033917L);
p.setDate(1, d);
d.setTime(0);
p.executeUpdate();
```

This code snippet inserts `Date(0)` in the database instead of `Date(1181676033917L)` because the bound values are not copied by JDBC driver implementation for performance reasons.

B

Oracle RAC Fast Application Notification

Oracle Database 12*c* Release 1 (12.1) introduces a new set of APIs for Oracle RAC Fast Application Notification (FAN) events. These APIs provide an alternative for taking advantage of the high-availability (HA) features of Oracle Database, if you do not use Universal Connection Pool or Oracle JDBC connection caching. These APIs are not a part of Oracle JDBC APIs.

This appendix covers the following topics:

- Overview of Oracle RAC Fast Application Notification
- Installing and Configuring Oracle RAC Fast Application Notification
- Using Oracle RAC Fast Application Notification
- Implementing a Connection Cache

This feature depends on the Oracle Notification System (ONS) message transport mechanism. This feature requires configuring your system, servers, and clients to use ONS.

For using Oracle RAC Fast Application Notification, the `simplefan.jar` file must be present in the classpath, and either the `ons.jar` file must be present in the classpath or an Oracle Notification Services (ONS) client must be installed and running in the client system.

Overview of Oracle RAC Fast Application Notification

The Oracle RAC Fast Application Notification (FAN) feature provides a simplified API for accessing FAN events through a callback mechanism. This mechanism enables application code to receive a callback when a FAN event occurs. These APIs are referred to as Oracle RAC FAN APIs in this appendix.

The Oracle RAC FAN APIs provide FAN event notification for developing more responsive applications that can take full advantage of Oracle Database HA features. If you do not want to use Universal Connection Pool, but want to work with FAN events implementing your own connection cache, then you should use Oracle RAC Fast Application Notification.

Note:

- If you do not want to implement your own connection cache, then you should use Universal Connection Pooling to get all the advantages of Oracle RAC Fast Application Notification, along with many additional benefits. For more information on Universal Connection Pooling, refer to *Oracle Universal Connection Pool for JDBC Developer's Guide*.
- Starting from Oracle Database 12*c* Release 1 (12.1), implicit connection cache is desupported. Oracle recommends to use Universal Connection Pool in place of implicit connection cache.

FAN event notifications sent by Oracle RAC. This is achieved by enabling the code to respond to FAN events in the following way:

- Listening for Oracle RAC service down and node down events
- Listening for load balancing advisory events and responding to them

This feature exposes a subset of FAN events, which are the notifications sent by a cluster running Oracle RAC, to inform the subscribers about the configuration changes within the cluster. The supported FAN events are the following:

- Service down

  The service down events notify that the managed resources are down and currently not available for access. There are two kinds of service down events:

  – Events indicating that a particular instance of a service is down and the instance is no longer able to accept work.

  – Events indicating that *all* instances of a service are down and the service is no longer able to accept work.

  When the last instance of a service goes down, then the subscriber receives both events. The events may or may not arrive together.

- Node down

  The node down events notify that the Oracle RAC node identified by the host identifier is down and not reachable. The cluster sends node down events when a node is no longer able to accept work.

- Load balancing advisory

  The load balancing advisory events provide metrics for load balancing algorithms. Load balancing advisories are sent regularly to inform subscribers of the recommended distribution of work among the available nodes.

Note: If you want to implement your own connection cache, only then you should use Oracle RAC Fast Application Notification. Otherwise, you should use Universal Connection Pooling to get all the advantages of Oracle RAC Fast Application Notification, along with many additional benefits. For more information on Universal Connection Pooling, refer to *Oracle Universal Connection Pool for JDBC Developer's Guide*.

Installing and Configuring Oracle RAC Fast Application Notification

You can install the Oracle RAC FAN APIs by performing the following steps:

1. Download the `simplefan.jar` file from the following link

   http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html

2. Add the `simplefan.jar` file to the classpath.
3. Perform the following in your Java code:
   a. Get an instance of the `FanManager` class by using the `getInstance` method.
   b. Configure the event daemon using the `configure` method of the `FanManager` class. The `configure` method sets the following properties:

      `onsNodes`: A comma separated list of `host:port` pairs of ONS daemons that the ONS runtime in this Java VM should communicate with. The `host` in a `host:port` pair is the host name of a system running the ONS daemon. The `port` is the local port configuration parameter for that daemon.

      `onsWalletFile`: The path name of the ONS wallet file. The wallet file is the path to a local wallet file used by SSL to store SSL certificates. Same as wallet file configuration parameter to ONS daemon.

      `onsWalletPassword`: The password for accessing the ONS wallet file.

For a detailed description of the Oracle RAC FAN APIs, refer to *Oracle Database RAC FAN Events Java API Reference*.

Using Oracle RAC Fast Application Notification

Example B–1 provides an example to use Oracle RAC Fast Application Notification in your code. This example code prints the event data to the standard output device.

This example code demonstrates sample usages of Oracle RAC FAN APIs by overloading the `handleFanEvent` method to accept different FAN event notifications as arguments. The example code also displays event data such as

- Name of the system sending the FAN event notification
- Timestamp of the FAN event notification
- Load status of the FAN event notification

Example B–1 Example of Sample Code Using Oracle RAC FAN API

```
...
...
Properties props = new Properties();
props.putProperty("serviceName", "gl");
FanSubscription sub = FanManager.getInstance().subscribe(props);
sub.addListener(new FanEventListener()) {
  public void handleFanEvent(ServiceDownEvent event) {
    try {
      System.out.println(event.getTimestamp());
      System.out.println(event.getServiceName());
      System.out.println(event.getDatabaseUniqueName());
      System.out.println(event.getReason());
      ServiceMemberEvent me = se.getServiceMemberEvent();
      if (me != null) {
        System.out.println(me.getInstanceName());
```

```
        System.out.println(me.getNodeName());
        System.out.println(me.getServiceMemberStatus());
      }
      ServiceCompositeEvent ce = se.getServiceCompositeEvent();
      if (ce != null) {
        System.out.println(ce.getServiceCompositeStatus());
      }
    }
    catch (Throwable t) {
      // handle all exceptions and errors
      t.printStackTrace(System.err);
    }
  }
  public void handleFanEvent(NodeDownEvent event) {
    try {
      System.out.println(event.getTimestamp());
      System.out.println(ne.getNodeName());
      System.out.println(ne.getIncarnation());
    }
    catch (Throwable t) {
      // handle all exceptions and errors
      t.printStackTrace(System.err);
    }
  }
  public void handleFanEvent(LoadAdvisoryEvent event) {
    try {
      System.out.println(event.getTimestamp());
      System.out.println(le.getServiceName());
      System.out.println(le.getDatabaseUniqueName());
      System.out.println(le.getInstanceName());
      System.out.println(le.getPercent());
      System.out.println(le.getServiceQuality());
      System.out.println(le.getLoadStatus());
    }
    catch (Throwable t) {
      // handle all exceptions and errors
      t.printStackTrace(System.err);
    }
  }
});
```

Implementing a Connection Cache

You must implement your own connection cache for using Oracle RAC FAN APIs. Consider the following points before you implement a connection cache using the Oracle RAC FAN APIs:

- Oracle RAC FAN APIs provide a minimal subset of FAN events.
- Oracle RAC FAN APIs support only ONS events. If you want your application to support corresponding supercluster events, then you may require additions to the subscription properties.
- Oracle RAC FAN APIs do not enable application code to send FAN events.

C

JDBC Coding Tips

This appendix describes methods to optimize a Java Database Connectivity (JDBC) application or applet. It includes the following topics:

- JDBC and Multithreading
- Performance Optimization of JDBC Programs
- Transaction Isolation Levels and Access Modes in JDBC

JDBC and Multithreading

Oracle JDBC drivers provide full support for, and are highly optimized for, applications that use Java multithreading. Controlled serial access to a connection, such as that provided by connection caching, is both necessary and encouraged. However, Oracle strongly discourages sharing a database connection among multiple threads. Avoid allowing multiple threads to access a connection simultaneously. If multiple threads must share a connection, use a disciplined begin-using/end-using protocol.

Keep the following points in mind while working on multithreaded applications:

- Use the `Connection` object as a local variable.
- Close the connection in the `finally` block before exiting the method. For example:

```
Connection conn = null;
try
{
    ...
}
finally
{
   if(conn != null) conn.close();
}
```

- Do not share `Connection` objects between threads.
- Never synchronize on JDBC objects because it is done internally by the driver.
- Use the `Statement.setQueryTimeout` method to set the time to execute a query instead of cancelling the long-running query from a different thread.
- Use the `Statement.cancel` method for SQL operations like `SELECT`, `UPDATE`, or `DELETE`.
- Use the `Connection.cancel` method for SQL operations like `COMMIT`, `ROLLBACK`, and so on.

- Do not use the `Thread.interrupt` method.

Performance Optimization of JDBC Programs

You can significantly enhance the performance of your JDBC programs by using any of these features:

- Disabling Auto-Commit Mode
- Standard Fetch Size and Oracle Row Prefetching
- Setting the Session Data Unit Size
- JDBC Update Batching
- Statement Caching
- Mapping Between Built-in SQL and Java Types

Disabling Auto-Commit Mode

Auto-commit mode indicates to the database whether to issue an automatic `COMMIT` operation after every SQL operation. Being in auto-commit mode can be expensive in terms of time and processing effort if, for example, you are repeating the same statement with different bind variables.

By default, new connection objects are in auto-commit mode. However, you can disable auto-commit mode with the `setAutoCommit` method of the connection object, either `java.sql.Conection` or `oracle.jdbc.OracleConnection`.

In auto-commit mode, the `COMMIT` operation occurs either when the statement completes or the next execute occurs, whichever comes first. In the case of statements returning a `ResultSet` object, the statement completes when the last row of the Result Set has been retrieved or when the Result Set has been closed. In more complex cases, a single statement can return multiple results as well as output parameter values. Here, the `COMMIT` occurs when all results and output parameter values have been retrieved.

If you disable auto-commit mode with a `setAutoCommit(false)` call, then you must manually commit or roll back groups of operations using the `commit` or `rollback` method of the connection object.

Example

The following example illustrates loading the driver and connecting to the database. Because new connections are in auto-commit mode by default, this example shows how to disable auto-commit. In the example, `conn` represents the `Connection` object, and `stmt` represents the `Statement` object.

```
// Connect to the database
// You can put a database host name after the @ sign in the connection URL.
   OracleDataSource ods = new OracleDataSource();
   ods.setURL("jdbc:oracle:oci:@");
   ods.setUser("HR");
   ods.setPassword("hr");
   Connection conn = ods.getConnection();

// It's faster when auto commit is off
conn.setAutoCommit (false);

// Create a Statement
Statement stmt = conn.createStatement ();
```

. . .

Standard Fetch Size and Oracle Row Prefetching

Oracle JDBC connection and statement objects allow you to specify the number of rows to prefetch into the client with each trip to the database while a result set is being populated during a query. You can set a value in a connection object that affects each statement produced through that connection, and you can override that value in any particular statement object. The default value in a connection object is 10. Prefetching data into the client reduces the number of round-trips to the server.

Similarly, and with more flexibility, JDBC 2.0 enables you to specify the number of rows to fetch with each trip, both for statement objects (affecting subsequent queries) and for result set objects (affecting row refetches). By default, a result set uses the value for the statement object that produced it. If you do not set the JDBC 2.0 fetch size, then the Oracle connection row-prefetch value is used by default.

> **See Also:** "Fetch Size" on page 17-4

Setting the Session Data Unit Size

Session data unit (SDU) is a buffer that Oracle Net uses to place data before transmitting it across the network. Oracle Net sends the data in the buffer either when the request is completed or when it is full.

You can configure the SDU and obtain the following benefits, among others:

- Reduction in the time required to transmit a SQL query and result across the network
- Transmission of larger chunks of data

> **Note:** The footprint of the client and the server process increase if you set a bigger SDU size.

> **See Also:** *Oracle Database Net Services Administrator's Guide*

This section describes the following:

- Setting the SDU Size for the Database Server
- Setting the SDU Size for JDBC Thin Client

Setting the SDU Size for the Database Server

To set the SDU size for the database server, configure the `DEFAULT_SDU_SIZE` parameter in the `sqlnet.ora` file.

> **See Also:** *Oracle Database Net Services Administrator's Guide* for more information about setting the SDU size for the database server

Setting the SDU Size for JDBC OCI Client

The JDBC OCI client uses Oracle Net layer. So, you can set the SDU size for the JDBC OCI client by configuring the `DEFAULT_SDU_SIZE` parameter in the `sqlnet.ora` file.

> **See Also:** *Oracle Database Net Services Administrator's Guide* for more information about setting the SDU size for the database server

Setting the SDU Size for JDBC Thin Client

You can set the SDU size for JDBC thin client by specifying it in the `DESCRIPTION` parameter for a particular connection descriptor.

```
sales.example.com=
(DESCRIPTION=
    (SDU=11280)
      (ADDRESS=(PROTOCOL=tcp)(HOST=sales-server)(PORT=5221))
    (CONNECT_DATA=
     (SERVICE_NAME=sales.example.com))
)
```

JDBC Update Batching

Oracle JDBC drivers enable you to accumulate `INSERT`, `DELETE`, and `UPDATE` operations of prepared statements at the client and send them to the server in batches. This feature reduces round-trips to the server.

Note: Oracle recommends to keep the batch sizes in the range of 100 or less. Larger batches provide little or no performance improvement and may actually reduce performance due to the client resources required to handle the large batch.

Statement Caching

Statement caching improves performance by caching executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. Applications use the statement cache to cache statements associated with a particular physical connection. When you enable Statement caching, a Statement object is cached when you call the `close` method. Because each physical connection has its own cache, multiple caches can exist if you enable Statement caching for multiple physical connections.

Note: The Oracle JDBC drivers are optimized for use with the Oracle Statement cache. Oracle strongly recommends that you use the Oracle Statement cache (implicit or explicit).

When you enable Statement caching on a connection cache, the logical connections benefit from the Statement caching that is enabled on the underlying physical connection. If you try to enable Statement caching on a logical connection held by a connection cache, then this will throw an exception.

See Also: Chapter 20, "Statement and Result Set Caching"

Mapping Between Built-in SQL and Java Types

The SQL built-in types are those types with system-defined names, such as `NUMBER`, and `CHAR`, as opposed to the Oracle objects, varray, and nested table types, which have user-defined names. In JDBC programs that access data of built-in SQL types, all type conversions are unambiguous, because the program context determines the Java type to which a SQL datum will be converted.

The most efficient way to access numeric data is to use primitive Java types like `int`, `float`, `long`, and `double`. However, the range of values of these types do not exactly

match the range of values of the SQL `NUMBER` data type. As a result, there may be some loss of information. If absolute precision is required across the entire value range, then use the `BigDecimal` type.

All character data is converted to the UCS2 character set of Java. The most efficient way to access character data is as `java.lang.String`. In worst case, this can cause a loss of information when two or more characters in the database character set map to a single UCS2 character. Since Oracle Database 11*g*, all characters in the character set map to the characters in the UCS2 character set. However, some characters do map to surrogate pairs.

Transaction Isolation Levels and Access Modes in JDBC

Read-only connections are supported by Oracle JDBC drivers, but not by the Oracle server.

For transactions, the Oracle server supports only the `TRANSACTION_READ_COMMITTED` and `TRANSACTION_SERIALIZABLE` transaction isolation levels. The default is `TRANSACTION_READ_COMMITTED`. Use the following methods of the `oracle.jdbc.OracleConnection` interface to get and set the level:

- `getTransactionIsolation`: Gets this connection's current transaction isolation level.
- `setTransactionIsolation`: Changes the transaction isolation level, using one of the `TRANSACTION_*` values.

D

JDBC Error Messages

This appendix briefly discusses the general structure of Java Database Connectivity (JDBC) error messages, then lists general JDBC error messages and TTC error messages that Oracle JDBC drivers can return. The appendix is organized as follows:

- General Structure of JDBC Error Messages
- General JDBC Messages
- Native XA Messages
- TTC Messages

Each of the message lists is first sorted by `ORA` number, and then alphabetically.

See Also: "Processing SQL Exceptions" on page 2-20

General Structure of JDBC Error Messages

The general JDBC error message structure allows run-time information to be appended to the end of a message, following a colon, as follows:

```
<error_message>:<extra_info>
```

For example, a "closed statement" error might be displayed as follows:

```
Closed Statement:next
```

This indicates that the exception was thrown during a call to the `next` method (of a result set object).

In some cases, the user can find the same information in a stack trace.

General JDBC Messages

This section lists general JDBC error messages, first sorted by the `ORA` number, and then in alphabetic order in the following subsections:

- JDBC Messages Sorted by ORA Number
- JDBC Messages Sorted in Alphabetic Order

Note: The `ORA-17033` and `ORA-17034` error messages use the term SQL92. The JDBC escape syntax was previously known as SQL92 Syntax or SQL92 escape syntax.

JDBC Messages Sorted by ORA Number

The following table lists the JDBC error messages sorted by the ORA number:

Table D–1 JDBC Messages Sorted by ORA Number

ORA Number	Message
ORA-17001	Internal Error
ORA-17002	Io exception
ORA-17003	Invalid column index
ORA-17004	Invalid column type
ORA-17005	Unsupported column type
ORA-17006	Invalid column name
ORA-17007	Invalid dynamic column
ORA-17008	Closed Connection
ORA-17009	Closed Statement
ORA-17010	Closed Resultset
ORA-17011	Exhausted Resultset
ORA-17012	Parameter Type Conflict
ORA-17014	ResultSet.next was not called
ORA-17015	Statement was cancelled
ORA-17016	Statement timed out
ORA-17017	Cursor already initialized
ORA-17018	Invalid cursor
ORA-17019	Can only describe a query
ORA-17020	Invalid row prefetch
ORA-17021	Missing defines
ORA-17022	Missing defines at index
ORA-17023	Unsupported feature
ORA-17024	No data read
ORA-17025	Error in defines.isNull ()
ORA-17026	Numeric Overflow
ORA-17027	Stream has already been closed
ORA-17028	Can not do new defines until the current ResultSet is closed
ORA-17029	setReadOnly: Read-only connections not supported
ORA-17030	READ_COMMITTED and SERIALIZABLE are the only valid transaction levels
ORA-17031	setAutoClose: Only support auto close mode on
ORA-17032	cannot set row prefetch to zero
ORA-17033	Malformed SQL92 string at position
ORA-17034	Non supported SQL92 token at position
ORA-17035	Character Set Not Supported !!

Table D–1 (Cont.) JDBC Messages Sorted by ORA Number

ORA Number	Message
ORA-17036	exception in OracleNumber
ORA-17037	Fail to convert between UTF8 and UCS2
ORA-17038	Byte array not long enough
ORA-17039	Char array not long enough
ORA-17040	Sub Protocol must be specified in connection URL
ORA-17041	Missing IN or OUT parameter at index:
ORA-17042	Invalid Batch Value
ORA-17043	Invalid stream maximum size
ORA-17044	Internal error: Data array not allocated
ORA-17045	Internal error: Attempt to access bind values beyond the batch value
ORA-17046	Internal error: Invalid index for data access
ORA-17047	Error in Type Descriptor parse
ORA-17048	Undefined type
ORA-17049	Inconsistent java and sql object types
ORA-17050	no such element in vector
ORA-17051	This API cannot be be used for non-UDT types
ORA-17052	This ref is not valid
ORA-17053	The size is not valid
ORA-17054	The LOB locator is not valid
ORA-17055	Invalid character encountered in
ORA-17056	Non supported character set (add orai18n.jar in your classpath)
ORA-17057	Closed LOB
ORA-17058	Internal error: Invalid NLS Conversion ratio
ORA-17059	Fail to convert to internal representation
ORA-17060	Fail to construct descriptor
ORA-17061	Missing descriptor
ORA-17062	Ref cursor is invalid
ORA-17063	Not in a transaction
ORA-17064	Invalid Sytnax or Database name is null
ORA-17065	Conversion class is null
ORA-17066	Access layer specific implementation needed
ORA-17067	Invalid Oracle URL specified
ORA-17068	Invalid argument(s) in call
ORA-17069	Use explicit XA call
ORA-17070	Data size bigger than max size for this type
ORA-17071	Exceeded maximum VARRAY limit

Table D–1 (Cont.) JDBC Messages Sorted by ORA Number

ORA Number	Message
ORA-17072	Inserted value too large for column
ORA-17074	invalid name pattern
ORA-17075	Invalid operation for forward only resultset
ORA-17076	Invalid operation for read only resultset
ORA-17077	Fail to set REF value
ORA-17078	Cannot do the operation as connections are already opened
ORA-17079	User credentials doesn't match the existing ones
ORA-17080	invalid batch command
ORA-17081	error occurred during batching
ORA-17082	No current row
ORA-17083	Not on the insert row
ORA-17084	Called on the insert row
ORA-17085	Value conflicts occurs
ORA-17086	Undefined column value on the insert row
ORA-17087	Ignored performance hint: setFetchDirection()
ORA-17088	Unsupported syntax for requested resultset type and concurrency level
ORA-17089	internal error
ORA-17090	operation not allowed
ORA-17091	Unable to create resultset at the requested type and/or concurrency level
ORA-17092	JDBC statements cannot be created or executed at end of call processing
ORA-17093	OCI operation returned OCI_SUCCESS_WITH_INFO
ORA-17094	Object type version mismatched
ORA-17095	Statement cache size has not been set
ORA-17096	Statement Caching cannot be enabled for this logical connection.
ORA-17097	Invalid PL/SQL Index Table element type
ORA-17098	Invalid empty lob operation
ORA-17099	Invalid PL/SQL Index Table array length
ORA-17100	Invalid database Java Object
ORA-17101	Invalid properties in OCI Connection Pool Object
ORA-17102	Bfile is read only
ORA-17103	invalid connection type to return via getConnection. Use getJavaSqlConnection instead
ORA-17104	SQL statement to execute cannot be empty or null
ORA-17105	connection session time zone was not set
ORA-17106	invalid JDBC-OCI driver connection pool configuration specified

Table D–1 (Cont.) JDBC Messages Sorted by ORA Number

ORA Number	Message
ORA-17107	invalid proxy type specified
ORA-17108	No max length specified in defineColumnType
ORA-17109	standard Java character encoding not found
ORA-17110	execution completed with warning
ORA-17111	Invalid connection cache TTL timeout specified
ORA-17112	Invalid thread interval specified
ORA-17113	Thread interval value is more than the cache timeout value
ORA-17114	could not use local transaction commit in a global transaction
ORA-17115	could not use local transaction rollback in a global transaction
ORA-17116	could not turn on auto-commit in an active global transaction
ORA-17117	could not set savepoint in an active global transaction
ORA-17118	could not obtain ID for a named Savepoint
ORA-17119	could not obtain name for an un-named Savepoint
ORA-17120	could not set a Savepoint with auto-commit on
ORA-17121	could not rollback to a Savepoint with auto-commit on
ORA-17122	could not rollback to a local txn Savepoint in a global transaction
ORA-17123	Invalid statement cache size specified
ORA-17124	Invalid connection cache Inactivity timeout specified
ORA-17125	Improper statement type returned by explicit cache
ORA-17126	Fixed Wait timeout elapsed
ORA-17127	Invalid Fixed Wait timeout specified
ORA-17128	SQL string is not Query
ORA-17129	SQL string is not a DML Statement
ORA-17132	Invalid conversion requested
ORA-17133	UNUSED
ORA-17134	Length of named parameter in SQL exceeded 32 characters
ORA-17135	Parameter name used in setXXXStream appears more than once in SQL
ORA-17136	Malformed DATALINK URL, try getString() instead
ORA-17137	Connection Caching Not Enabled or Not a Valid Cache Enabled DataSource
ORA-17138	Invalid Connection Cache Name. Must be a valid String and Unique
ORA-17139	Invalid Connection Cache Properties
ORA-17140	Connection Cache with this Cache Name already exists
ORA-17141	Connection Cache with this Cache Name does not exist
ORA-17142	Connection Cache with this Cache Name is Disabled
ORA-17143	Invalid or Stale Connection found in the Connection Cache

Table D–1 (Cont.) JDBC Messages Sorted by ORA Number

ORA Number	Message
ORA-17144	statement handle not executed
ORA-17145	Invalid ONS Event received
ORA-17146	Invalid ONS Event Version received
ORA-17147	Attempt to set a parameter name that does not occur in the SQL
ORA-17148	Method only implemented in thin
ORA-17149	This is already a proxy session
ORA-17150	Wrong arguments for proxy session
ORA-17151	Clob is too large to be stored in a Java String
ORA-17152	This method is only implemented in logical connections
ORA-17153	This method is only implemented in physical connections
ORA-17154	Cannot map Oracle character to Unicode
ORA-17155	Cannot map Unicode to Oracle character
ORA-17156	Invalid array size for End-to-End metrics values
ORA-17157	setString can only process strings of less than 32766 chararacters
ORA-17158	duration is invalid for this function
ORA-17159	metric value for end-to-end tracing is too long
ORA-17160	execution context id sequence number out of range
ORA-17161	Invalid transaction mode used
ORA-17162	Unsupported holdability value
ORA-17163	Can not use getXAConnection() when connection caching is enabled
ORA-17164	Can not call getXAResource() from physical connection with caching on
ORA-17165	DBMS_JDBC package not preset in server for this connection
ORA-17166	Cannot perform fetch on a PLSQL statement
ORA-17167	PKI classes not found. To use 'connect /' functionality, oraclepki.jar must be in the classpath
ORA-17168	encountered a problem with the Secret Store. Check the wallet location for the presence of an open wallet (cwallet.sso) and ensure that this wallet contains the correct credentials using the mkstore utility
ORA-17169	Cannot bind stream to a ScrollableResultSet or UpdatableResultSet
ORA-17170	The Namespace cannot be empty
ORA-17171	The attribute length cannot exceed 30 chars
ORA-17172	That value of the attribute cannot exceed 400 chars
ORA-17173	Not all return parameters registered
ORA-17174	The only supported namespace is CLIENTCONTEXT
ORA-17175	Error during remote ONS configuration
ORA-17259	SQLXML cannot find the XML support jar file in the classpath

Table D–1 (Cont.) JDBC Messages Sorted by ORA Number

ORA Number	Message
ORA-17260	Attempt to read an empty SQLXML
ORA-17261	Attempt to read a SQLXML that is not readable
ORA-17262	Attempt to write a SQLXML that is not writeable
ORA-17263	SQLXML cannot create a Result of that type
ORA_17264	SQLXML cannoct create a Source of that type

JDBC Messages Sorted in Alphabetic Order

The following table lists the JDBC error messages sorted in alphabetic order:

Table D–2 JDBC Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17066	Access layer specific implementation needed
ORA-17261	Attempt to read a SQLXML that is not readable
ORA-17260	Attempt to read an empty SQLXML
ORA-17147	Attempt to set a parameter name that does not occur in the SQL
ORA-17262	Attempt to write a SQLXML that is not writeable
ORA-17102	Bfile is read only
ORA-17038	Byte array not long enough
ORA-17084	Called on the insert row
ORA-17164	Can not call getXAResource() from physical connection with caching on
ORA-17028	Can not do new defines until the current ResultSet is closed
ORA-17163	Can not use getXAConnection() when connection caching is enabled
ORA-17019	Can only describe a query
ORA-17169	Cannot bind stream to a ScrollableResultSet or UpdatableResultSet
ORA-17078	Cannot do the operation as connections are already opened
ORA-17154	Cannot map Oracle character to Unicode
ORA-17155	Cannot map Unicode to Oracle character
ORA-17166	Cannot perform fetch on a PLSQL statement
ORA-17032	Cannot set row prefetch to zero
ORA-17039	Char array not long enough
ORA-17035	Character Set Not Supported !!
ORA-17151	Clob is too large to be stored in a Java String
ORA-17008	Closed Connection
ORA-17057	Closed LOB
ORA-17010	Closed Resultset
ORA-17009	Closed Statement

Table D–2 (Cont.) JDBC Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17140	Connection Cache with this Cache Name already exists
ORA-17141	Connection Cache with this Cache Name does not exist
ORA-17142	Connection Cache with this Cache Name is Disabled
ORA-17137	Connection Caching Not Enabled or Not a Valid Cache Enabled DataSource
ORA-17105	Connection session time zone was not set
ORA-17065	Conversion class is null
ORA-17118	Could not obtain ID for a named Savepoint
ORA-17119	Could not obtain name for an un-named Savepoint
ORA-17122	Could not rollback to a local txn Savepoint in a global transaction
ORA-17121	Could not rollback to a Savepoint with auto-commit on
ORA-17120	Could not set a Savepoint with auto-commit on
ORA-17117	Could not set savepoint in an active global transaction
ORA-17116	Could not turn on auto-commit in an active global transaction
ORA-17114	Could not use local transaction commit in a global transaction
ORA-17115	Could not use local transaction rollback in a global transaction
ORA-17017	Cursor already initialized
ORA-17070	Data size bigger than max size for this type
ORA-17165	DBMS_JDBC package not preset in server for this connection
ORA-17158	Duration is invalid for this function
ORA-17168	Encountered a problem with the Secret Store. Check the wallet location for the presence of an open wallet (cwallet.sso) and ensure that this wallet contains the correct credentials using the mkstore utility
ORA-17175	Error during remote ONS configuration
ORA-17025	Error in defines.isNull ()
ORA-17047	Error in Type Descriptor parse
ORA-17081	error occurred during batching
ORA-17071	Exceeded maximum VARRAY limit
ORA-17036	Exception in OracleNumber
ORA-17110	Execution completed with warning
ORA-17160	Execution context id sequence number out of range
ORA-17011	Exhausted Resultset
ORA-17060	Fail to construct descriptor
ORA-17037	Fail to convert between UTF8 and UCS2
ORA-17059	Fail to convert to internal representation
ORA-17077	Fail to set REF value
ORA-17126	Fixed Wait timeout elapsed

Table D–2 (Cont.) JDBC Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17087	Ignored performance hint: setFetchDirection()
ORA-17125	Improper statement type returned by explicit cache
ORA-17049	Inconsistent java and sql object types
ORA-17072	Inserted value too large for column
ORA-17001	Internal Error
ORA-17089	internal error
ORA-17045	Internal error: Attempt to access bind values beyond the batch value
ORA-17044	Internal error: Data array not allocated
ORA-17046	Internal error: Invalid index for data access
ORA-17058	Internal error: Invalid NLS Conversion ratio
ORA-17068	Invalid argument(s) in call
ORA-17156	Invalid array size for End-to-End metrics values
ORA-17080	invalid batch command
ORA-17042	Invalid Batch Value
ORA-17055	Invalid character encountered in
ORA-17003	Invalid column index
ORA-17006	Invalid column name
ORA-17004	Invalid column type
ORA-17124	Invalid connection cache Inactivity timeout specified
ORA-17138	Invalid Connection Cache Name. Must be a valid String and Unique
ORA-17139	Invalid Connection Cache Properties
ORA-17111	Invalid connection cache TTL timeout specified
ORA-17103	invalid connection type to return via getConnection. Use getJavaSqlConnection instead
ORA-17132	Invalid conversion requested
ORA-17018	Invalid cursor
ORA-17100	Invalid database Java Object
ORA-17007	Invalid dynamic column
ORA-17098	Invalid empty lob operation
ORA-17127	Invalid Fixed Wait timeout specified
ORA-17106	invalid JDBC-OCI driver connection pool configuration specified
ORA-17074	invalid name pattern
ORA-17145	Invalid ONS Event received
ORA-17146	Invalid ONS Event Version received
ORA-17075	Invalid operation for forward only resultset
ORA-17076	Invalid operation for read only resultset

Table D–2 (Cont.) JDBC Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17143	Invalid or Stale Connection found in the Connection Cache
ORA-17067	Invalid Oracle URL specified
ORA-17099	Invalid PL/SQL Index Table array length
ORA-17097	Invalid PL/SQL Index Table element type
ORA-17101	Invalid properties in OCI Connection Pool Object
ORA-17107	invalid proxy type specified
ORA-17020	Invalid row prefetch
ORA-17123	Invalid statement cache size specified
ORA-17043	Invalid stream maximum size
ORA-17064	Invalid Sytnax or Database name is null
ORA-17112	Invalid thread interval specified
ORA-17161	Invalid transaction mode used
ORA-17002	Io exception
ORA-17092	JDBC statements cannot be created or executed at end of call processing
ORA-17134	Length of named parameter in SQL exceeded 32 characters
ORA-17136	Malformed DATALINK URL, try getString() instead
ORA-17033	Malformed SQL92 string at position
ORA-17148	Method only implemented in thin
ORA-17159	metric value for end-to-end tracing is too long
ORA-17021	Missing defines
ORA-17022	Missing defines at index
ORA-17061	Missing descriptor
ORA-17041	Missing IN or OUT parameter at index:
ORA-17082	No current row
ORA-17024	No data read
ORA-17108	No max length specified in defineColumnType
ORA-17050	no such element in vector
ORA-17056	Non supported character set (add orai18n.jar in your classpath)
ORA-17034	Non supported SQL92 token at position
ORA-17173	Not all return parameters registered
ORA-17063	Not in a transaction
ORA-17083	Not on the insert row
ORA-17026	Numeric Overflow
ORA-17094	Object type version mismatched
ORA-17093	OCI operation returned OCI_SUCCESS_WITH_INFO
ORA-17090	operation not allowed

Table D–2 (Cont.) JDBC Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17135	Parameter name used in setXXXStream appears more than once in SQL
ORA-17012	Parameter Type Conflict
ORA-17167	PKI classes not found. To use 'connect /' functionality, oraclepki.jar must be in the classpath
ORA-17030	READ_COMMITTED and SERIALIZABLE are the only valid transaction levels
ORA-17062	Ref cursor is invalid
ORA-17014	ResultSet.next was not called
ORA-17031	setAutoClose: Only support auto close mode on
ORA-17029	setReadOnly: Read-only connections not supported
ORA-17157	setString can only process strings of less than 32766 chararacters
ORA-17104	SQL statement to execute cannot be empty or null
ORA-17129	SQL string is not a DML Statement
ORA-17128	SQL string is not Query
ORA-17263	SQLXML cannot create a Result of that type
ORA_17264	SQLXML cannoct create a Source of that type
ORA-17259	SQLXML cannot find the XML support jar file in the classpath
ORA-17109	standard Java character encoding not found
ORA-17095	Statement cache size has not been set
ORA-17096	Statement Caching cannot be enabled for this logical connection.
ORA-17144	statement handle not executed
ORA-17016	Statement timed out
ORA-17015	Statement was cancelled
ORA-17027	Stream has already been closed
ORA-17040	Sub Protocol must be specified in connection URL
ORA-17172	That value of the attribute cannot exceed 400 chars
ORA-17171	The attribute length cannot exceed 30 chars
ORA-17054	The LOB locator is not valid
ORA-17170	The Namespace cannot be empty
ORA-17174	The only supported namespace is CLIENTCONTEXT
ORA-17053	The size is not valid
ORA-17051	This API cannot be be used for non-UDT types
ORA-17149	This is already a proxy session
ORA-17152	This method is only implemented in logical connections
ORA-17153	This method is only implemented in physical connections
ORA-17052	This ref is not valid
ORA-17113	Thread interval value is more than the cache timeout value

Table D–2 (Cont.) JDBC Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17091	Unable to create resultset at the requested type and/or concurrency level
ORA-17086	Undefined column value on the insert row
ORA-17048	Undefined type
ORA-17005	Unsupported column type
ORA-17023	Unsupported feature
ORA-17162	Unsupported holdability value
ORA-17088	Unsupported syntax for requested resultset type and concurrency level
ORA-17133	UNUSED
ORA-17069	Use explicit XA call
ORA-17079	User credentials doesn't match the existing ones
ORA-17085	Value conflicts occurs
ORA-17150	Wrong arguments for proxy session

Native XA Messages

The following sections cover the JDBC error messages that are specific to the Native XA feature:

- Native XA Messages Sorted by ORA Number
- Native XA Messages Sorted in Alphabetic Order

Native XA Messages Sorted by ORA Number

The following table lists the Native XA messages sorted by the `ORA` number:

Table D–3 Native XA Messages Sorted by ORA Number

ORA Number	Message
ORA-17200	Unable to properly convert XA open string from Java to C
ORA-17201	Unable to properly convert XA close string from Java to C
ORA-17202	Unable to properly convert RM name from Java to C
ORA-17203	Could not casting pointer type to jlong
ORA-17204	Input array too short to hold OCI handles
ORA-17205	Failed to obtain OCISvcCtx handle from C-XA using xaoSvcCtx
ORA-17206	Failed to obtain OCIEnv handle from C-XA using xaoEnv
ORA-17207	The tnsEntry property was not set in DataSource
ORA-17213	C-XA returned XAER_RMERR during xa_open
ORA-17215	C-XA returned XAER_INVAL during xa_open
ORA-17216	C-XA returned XAER_PROTO during xa_open
ORA-17233	C-XA returned XAER_RMERR during xa_close
ORA-17235	C-XA returned XAER_INVAL during xa_close

Table D–3 (Cont.) Native XA Messages Sorted by ORA Number

ORA Number	Message
ORA-17236	C-XA returned XAER_PROTO during xa_close

Native XA Messages Sorted in Alphabetic Order

The following table lists the Native XA messages sorted in the alphabetic order:

Table D–4 Native XA Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17203	Could not casting pointer type to jlong
ORA-17235	C-XA returned XAER_INVAL during xa_close
ORA-17215	C-XA returned XAER_INVAL during xa_open
ORA-17236	C-XA returned XAER_PROTO during xa_close
ORA-17216	C-XA returned XAER_PROTO during xa_open
ORA-17233	C-XA returned XAER_RMERR during xa_close
ORA-17213	C-XA returned XAER_RMERR during xa_open
ORA-17206	Failed to obtain OCIEnv handle from C-XA using xaoEnv
ORA-17205	Failed to obtain OCISvcCtx handle from C-XA using xaoSvcCtx
ORA-17204	Input array too short to hold OCI handles
ORA-17207	The tnsEntry property was not set in DataSource
ORA-17202	Unable to properly convert RM name from Java to C
ORA-17201	Unable to properly convert XA close string from Java to C
ORA-17200	Unable to properly convert XA open string from Java to C

TTC Messages

This section lists TTC error messages, first sorted by the `ORA` number and then in alphabetic order in the following subsections:

- TTC Messages Sorted by ORA Number
- TTC Messages Sorted in Alphabetic Order

TTC Messages Sorted by ORA Number

The following table lists the TTC messages sorted by the `ORA` number:

Table D–5 TTC Messages Sorted by ORA Number

ORA Number	Message
ORA-17401	Protocol violation
ORA-17402	Only one RPA message is expected
ORA-17403	Only one RXH message is expected
ORA-17404	Received more RXDs than expected
ORA-17405	UAC length is not zero
ORA-17406	Exceeding maximum buffer length

Table D–5 (Cont.) TTC Messages Sorted by ORA Number

ORA Number	Message
ORA-17407	invalid Type Representation(setRep)
ORA-17408	invalid Type Representation(getRep)
ORA-17409	invalid buffer length
ORA-17410	No more data to read from socket
ORA-17411	Data Type representations mismatch
ORA-17412	Bigger type length than Maximum
ORA-17413	Exceding key size
ORA-17414	Insufficient Buffer size to store Columns Names
ORA-17415	This type hasn't been handled
ORA-17416	FATAL
ORA-17417	NLS Problem, failed to decode column names
ORA-17418	Internal structure's field length error
ORA-17419	Invalid number of columns returned
ORA-17420	Oracle Version not defined
ORA-17421	Types or Connection not defined
ORA-17422	Invalid class in factory
ORA-17423	Using a PLSQL block without an IOV defined
ORA-17424	Attempting different marshaling operation
ORA-17425	Returning a stream in PLSQL block
ORA-17426	Both IN and OUT binds are NULL
ORA-17427	Using Uninitialized OAC
ORA-17428	Logon must be called after connect
ORA-17429	Must be at least connected to server
ORA-17430	Must be logged on to server
ORA-17431	SQL Statement to parse is null
ORA-17432	invalid options in all7
ORA-17433	invalid arguments in call
ORA-17434	not in streaming mode
ORA-17435	invalid number of in_out_binds in IOV
ORA-17436	invalid number of outbinds
ORA-17437	Error in PLSQL block IN/OUT argument(s)
ORA-17438	Internal - Unexpected value
ORA-17439	Invalid SQL type
ORA-17440	DBItem/DBType is null
ORA-17441	Oracle Version not supported. Minimum supported version is 7.2.3.
ORA-17442	Refcursor value is invalid

Table D–5 (Cont.) TTC Messages Sorted by ORA Number

ORA Number	Message
ORA-17443	Null user or password not supported in THIN driver
ORA-17444	TTC Protocol version received from server not supported
ORA-17445	LOB already opened in the same transaction
ORA-17446	LOB already closed in the same transaction
ORA-17447	OALL8 is in an inconsistent state

TTC Messages Sorted in Alphabetic Order

The following table lists the TTC messages in the alphabetic order:

Table D–6 TTC Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17424	Attempting different marshaling operation
ORA-17412	Bigger type length than Maximum
ORA-17426	Both IN and OUT binds are NULL
ORA-17411	Data Type representations mismatch
ORA-17440	DBItem/DBType is null
ORA-17437	Error in PLSQL block IN/OUT argument(s)
ORA-17413	Exceding key size
ORA-17406	Exceeding maximum buffer length
ORA-17416	FATAL
ORA-17414	Insufficient Buffer size to store Columns Names
ORA-17438	Internal - Unexpected value
ORA-17418	Internal structure's field length error
ORA-17433	invalid arguments in call
ORA-17409	invalid buffer length
ORA-17422	Invalid class in factory
ORA-17419	Invalid number of columns returned
ORA-17435	invalid number of in_out_binds in IOV
ORA-17436	invalid number of outbinds
ORA-17432	invalid options in all7
ORA-17439	Invalid SQL type
ORA-17408	invalid Type Representation(getRep)
ORA-17407	invalid Type Representation(setRep)
ORA-17446	LOB already closed in the same transaction
ORA-17445	LOB already opened in the same transaction
ORA-17428	Logon must be called after connect
ORA-17429	Must be at least connected to server
ORA-17430	Must be logged on to server

Table D–6 (Cont.) TTC Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17417	NLS Problem, failed to decode column names
ORA-17410	No more data to read from socket
ORA-17434	not in streaming mode
ORA-17443	Null user or password not supported in THIN driver
ORA-17447	OALL8 is in an inconsistent state
ORA-17402	Only one RPA message is expected
ORA-17403	Only one RXH message is expected
ORA-17420	Oracle Version not defined
ORA-17441	Oracle Version not supported. Minimum supported version is 7.2.3.
ORA-17401	Protocol violation
ORA-17404	Received more RXDs than expected
ORA-17442	Refcursor value is invalid
ORA-17425	Returning a stream in PLSQL block
ORA-17431	SQL Statement to parse is null
ORA-17415	This type hasn't been handled
ORA-17444	TTC Protocol version received from server not supported
ORA-17421	Types or Connection not defined
ORA-17405	UAC length is not zero
ORA-17423	Using a PLSQL block without an IOV defined
ORA-17427	Using Uninitialized OAC

E

Troubleshooting

This appendix describes how to troubleshoot a Java Database Connectivity (JDBC) application or applet, and contains the following topics:

Common Problems

This section describes some common problems that you might encounter while using Oracle JDBC drivers. These problems include:

Memory Consumption for CHAR Columns Defined as OUT or IN/OUT Variables

In PL/SQL, when a `CHAR` or a `VARCHAR2` column is defined as a `OUT` or `IN/OUT` variable, the driver allocates a `CHAR` array of 32512 chars. This can cause a memory consumption problem. JDBC Thin driver does not allocate memory when using `VARCHAR2` output type. But JDBC OCI driver allocates memory for both `CHAR` and `VARCHAR2` types. So, CPU load in OCI driver is higher than Thin driver.

At previous releases, the solution to the problem was to invoke the `Statement.setMaxFieldSize` method. A better solution is to use `OracleCallableStatement.registerOutParameter`. Oracle encourages you always to call `registerOutParameter (int paramIndex, int sqlType, int scale, int maxLength)` on each `CHAR` or `VARCHAR2` column. This method is defined in `oracle.jdbc.OracleCallableStatement`. Use the fourth argument, `maxLength`, to limit the memory consumption. This parameter tells the driver how many characters are necessary to store this column. The column is truncated if the character array cannot hold the column data. The third argument, `scale`, is ignored by the driver.

Memory Leaks and Running Out of Cursors

If you receive messages that you are running out of cursors or that you are running out of memory, make sure that all your `Statement` and `ResultSet` objects are explicitly closed. Oracle JDBC drivers do not have finalizer methods. They perform cleanup routines by using the `close` method of the `ResultSet` and `Statement` classes. If you do not explicitly close your result set and statement objects, significant memory leaks can occur. You could also run out of cursors in the database. Closing a statement releases the corresponding cursor in the database.

Similarly, you must explicitly close `Connection` objects to avoid leaking and running out of cursors on the server-side. When you close the connection, the JDBC driver closes any open statement objects associated with it, thus releasing the cursor on the server-side.

Opening More than 16 OCI Connections for a Process

You may find that you are unable to open more than approximately 16 JDBC-OCI connections for a process at any given time. The most likely reasons for this would be either that the number of processes on the server exceeded the limit specified in the initialization file, or that the per-process file descriptors limit was exceeded. It is important to note that one JDBC-OCI connection can use more than one file descriptor (it might use anywhere between 3 and 4 file descriptors).

If the server allows more than 16 processes, then the problem could be with the per-process file descriptor limit. The possible solution would be to increase this limit.

Using statement.cancel

The JDBC standard method `Statement.cancel` attempts to cleanly stop the execution of a SQL statement by sending a message to the database. In response, the database stops execution and replies with an error message. The Java thread that invoked `Statement.execute` waits on the server, and continues execution only when it receives the error reply message invoked by the call of the other thread to `Statement.cancel` method.

As a result, the `Statement.cancel` method relies on the correct functioning of the network and the database. If either the network connection is broken or the database server is hung, the client does not receive the error reply to the cancel message. Frequently, when the server process dies, JDBC receives an `IOException` that frees the thread that invoked `Statement.execute`. In some circumstances, the server is hung, but JDBC does not receive an `IOException`. The `Statement.cancel` method does not free the thread that initiated the `Statement.execute` method.

Note: Remember the following points while working with the `Statement.cancel` method:

- Distinguish between Connection-level and Statement-level cancel. If a nonstatement execution, for example, a `ROLLBACK` is cancelled by a `statement.cancel` method, then we replay the command (only if it is `ROLLBACK`, `COMMIT`, `autoCommit ON`, `autoCommit OFF`, `VERSION`). To guarantee data integrity, we do not replay statement executions.
- Synchronize statement execution and statement cancel, so that the execution does not return until the cancel call is sent to the Database. This provides a better chance for the executing statement to be cancelled.
- Synchronize cancel calls, so that any new cancel request is ignored until the cancel in progress has completed the full protocol, that is, after the database receives an interrupt, act on it, and notify JDBC.

When JDBC does not receive an `IOException`, Oracle Net may eventually time out and close the connection. This causes an `IOException` and frees the thread. This process can take many minutes. For information about how to control this time-out, see the description of the `readTimeout` property for `OracleDatasource.setConnectionProperties`. You can also tune this time-out with certain Oracle Net settings.

See Also: *Oracle Database Net Services Administrator's Guide* for more information

The JDBC standard method `Statement.setQueryTimeout` relies on the `Statement.cancel` method. If execution continues longer than the specified time-out interval, then the monitor thread calls the `Statement.cancel` method. This is subject to all the same limitations described previously. As a result, there are cases when the time-out does not free the thread that invoked the `Statement.execute` method.

The length of time between execution and cancellation is not precise. This interval is no less than the specified time-out interval but can be several seconds longer. If the application has active threads running at high priority, then the interval can be arbitrarily longer. The monitor thread runs at high priority, but other high priority threads may keep it from running indefinitely. Note that the monitor thread is started only if there are statements executed with non zero time-out. There is only one monitor thread that monitors all Oracle JDBC statement execution.

Note: The `Statement.cancel` method and the `Statement.setQueryTimeout` method are not supported in the server-side internal driver. The server-side internal driver runs in the single-threaded server process and the Oracle JVM implements Java threads within this single-threaded process. If the server-side internal driver is executing a SQL statement, then no Java thread can call the `Statement.cancel` method. This also applies to the Oracle JDBC monitor thread.

Using JDBC with Firewalls

Firewall timeout for idle-connections may sever a connection. This can cause JDBC applications to hang while waiting for a connection. You can perform one or more of the following actions to avoid connections from being severed due to firewall timeout:

- If you are using connection caching or connection pooling, then always set the inactivity timeout value on the connection cache to be shorter than the firewall idle timeout value.
- Pass `oracle.jdbc.ReadTimeout` as connection property to enable read timeout on socket. The timeout value is in milliseconds.
- For both JDBC OCI and JDBC Thin drivers, use net descriptor to connect to the database and specify the `ENABLE=BROKEN` parameter in the `DESCRIPTION` clause in the connect descriptor. Also, set a lower value for `TCP_KEEPALIVE_INTERVAL`.
- Enable Oracle Net DCD by setting `SQLNET.EXPIRE_TIME=1` in the `sqlnet.ora` file on the server-side.

Frequent Abrupt Disconnection from Server

If the network is not reliable, then it is difficult for a client to detect the frequent disconnections when the server is abruptly disconnected. By default, a client running on Linux takes 7200 seconds (2 hours) to sense the abrupt disconnections. This value is equal to the value of the `tcp_keepalive_time` property. If you want your application to detect the disconnections faster, then you must set the value of the `tcp_keepalive_time`, `tcp_keepalive_interval`, and `tcp_keepalive_probes` properties to a lower value at the operating system level.

> **Note:** Setting a low value for the `tcp_keepalive_interval` property leads to frequent probe packets on the network, which can make the system slower. So, the value of this property should be set appropriately based on the system requirements.

Also, you must specify the `ENABLE=BROKEN` parameter in the `DESCRIPTION` clause in the connection descriptor. For example:

```
jdbc:oracle:thin:@(DESCRIPTION=(ENABLE=BROKEN)(ADDRESS=(PROTOCOL=tcp)(PORT=5221)(H
OST=myhost))(CONNECT_DATA=(SERVICE_NAME=orcl)))
```

Network Adapter Cannot Establish Connection

You may receive the following error while trying to establish a connection from a JDBC application to an Oracle instance:

```
java.sql.SQLException: Io exception:
  The Network Adapter could not establish connection

SQLException: SQLState (null) vendor code (17002)
```

This error may occur even if all or any of the following conditions is true:

- You are able to establish a SQL*Plus connection from the same client to the same Oracle instance.
- You are able to establish a JDBC OCI connection, but not a JDBC Thin connection from the same client to the same Oracle instance.

- The same JDBC application is able to connect from a different client to the same Oracle instance.
- The same behavior applies whether the initial JDBC connection string specifies a host name or an IP address.

One or more of the following reasons can cause this error:

- The host name to which you are trying to establish the connection is incorrect.
- The port number you are using to establish the connection is wrong.
- The NIC card supports both IPv4 and IPv6.
- The Oracle instance is configured for MTS, but the JDBC connection uses a shared server instead of a dedicated server.

You can quickly diagnose these above-mentioned reasons by using SQL*Plus, except for the issue with the NIC card. The following sections specify how to resolve this error, and also contains a sample application:

- Oracle Instance Configured with MTS Server Uses Shared Server
- JDBC Thin Driver with NIC Card Supporting Both IPv4 and IPv6
- Sample Application

Oracle Instance Configured with MTS Server Uses Shared Server

For resolving this error, you must verify whether the Oracle instance is configured for Multi-threaded Server (MTS) or not. If the Oracle instance is not configured for MTS, then it must be configured.

If the Oracle instance is configured for MTS, then you must force the JDBC connection to use a dedicated server instead of a shared server. You can achieve this by reconfiguring the server to use dedicated connections only. If it is not feasible to configure your server to use only dedicated connections, then you perform the following steps to set it from the client side:

For JDBC OCI Client

1. Add the `(SERVER=DEDICATED)` property to the TNS connection string stored in the `tnsnames.ora` file on the client.
2. Set the `USER_DEDICATED_SERVER=ON` in the `sqlnet.ora` file on the client.

For JDBC Thin:

You must specify a full name-value pair connection string (the same as it may appear in the `tnsnames.ora` file) instead of the short JDBC Thin syntax. For example, instead of the "`jdbc:oracle:thin:@host:port:sid`" connection string, you must use a connection string of the following form:

```
"jdbc:oracle:thin:@(DESCRIPTION="                  +
                      "(ADDRESS_LIST="             +
                          "(ADDRESS=(PROTOCOL=TCP)" +
                                   "(HOST=host)"   +
                                   "(PORT=port)"   +
                          ")"                      +
                      ")"                          +
                      "(CONNECT_DATA="             +
                          "(SERVICE_NAME=sid)"     +
                          "(SERVER=DEDICATED)"     +
                      ")"                          +
```

```
")"
```

JDBC Thin Driver with NIC Card Supporting Both IPv4 and IPv6

If the Network Interface Controller (NIC) card of the server is configured to support both IPv4 and IPv6, then some services may start with IPv6. Any client application that tries to connect using IPv4 to the service that is running with IPv6 (or the other way round) receives a connection refused error. If a JDBC thin client application tries to connect to the Database server, then the application may stop responding or fail with the following error:

```
java.sql.SQLException: Io exception: The Network Adapter could not establish the
connection Error Code: 17002
```

Use any of the following solutions to resolve this error:

- Indicate the Java Virtual Machine (JVM) to use IP protocol version 4. Launch the JVM, where the JDBC application is running, with the `-Djava.net.preferIPv4Stack` parameter as `true`. For example, suppose you are running a JDBC application named `jdbcTest`. Then execute the application in the following way:

  ```
  java -Djava.net.preferIPv4Stack=true jdbcTest
  ```

- Use the OCI JDBC driver.

Sample Application

Example E–1 shows a basic JDBC program that connects to a Database and can be used to test your connection. It enables to try all forms of connection using Oracle JDBC drivers.

Example E–1 Basic JDBC Program to Connect to a Database in Five Different Ways

```
import java.sql.*;
public class Jdbctest
{
   public static void main (String args[])
   {
         try
         {
                   /* Uncomment the next line for more connection information */
                   // DriverManager.setLogStream(System.out);
                   /* Set the host, port, and sid below to match the entries in
the listener.ora */
                   String host = "myhost.oracle.com";
                   String port = "5221";
                   String sid  = "orcl";
                  // or pass on command line arguments for all three items
                   if ( args.length >= 3 )
                   {
                             host = args[0];
                             port = args[1];
                             sid  = args[2];
                   }

                   String s1 = "jdbc:oracle:thin:@" + host + ":" + port + ":" +
sid ;
                   if ( args.length == 1 )
                   {
```

```
                                s1 =  "jdbc:oracle:oci8:@" + args[0];
                    }
                    if ( args.length == 4 )
                    {
                                s1 = "jdbc:oracle:" + args[3] + ":@" +
                                     "(description=(address=(host=" + host+
")(protocol=tcp)(port=" + port+ "))(connect_data=(sid="+ sid + ")))";
                    }
                    System.out.println( "Connecting with: " );
                    System.out.println( s1 );
                    DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
               Connection conn = DriverManager.getConnection( s1,"hr","hr");
                    DatabaseMetaData dmd = conn.getMetaData();

System.out.println("DriverVersion:["+dmd.getDriverVersion()+"]");
                    System.out.println("DriverMajorVersion:
["+dmd.getDriverMajorVersion()+"]");
                    System.out.println("DriverMinorVersion:
["+dmd.getDriverMinorVersion()+"]");
                    System.out.println("DriverName:["+dmd.getDriverName()+"]");
                    if ( conn!=null )
                       conn.close();
                    System.out.println("Done.");
          }
          catch ( SQLException e )
          {
                    System.out.println ("\n*** Java Stack Trace ***\n");
                    e.printStackTrace();
                   System.out.println ("\n*** SQLException caught ***\n");
                    while ( e != null )
                    {
                                System.out.println ("SQLState: " + e.getSQLState
());
                                System.out.println ("Message:  " + e.getMessage
());
                                System.out.println ("Error Code:   " +
e.getErrorCode ());
                                e = e.getNextException ();
                                System.out.println ("");
                     }
             }
  }
}
```

Basic Debugging Procedures

This section describes strategies for debugging a JDBC program:

- Oracle Net Tracing to Trap Network Events
- Third Party Debugging Tools

For information about processing SQL exceptions, including printing stack traces to aid in debugging, see "Processing SQL Exceptions" on page 2-20.

Oracle Net Tracing to Trap Network Events

You can enable client and server Oracle-Net trace to trap the packets sent over Oracle Net. You can use client-side tracing only for the JDBC OCI driver; it is not supported for the JDBC Thin driver. You can find more information about tracing and reading trace files in the *Oracle Net Services Administrator's Guide*.

The trace facility produces a detailed sequence of statements that describe network events as they execute. "Tracing" an operation lets you obtain more information about the internal operations of the event. This information is printed to a readable file that identifies the events that led to the error. Several Oracle Net parameters in the `SQLNET.ORA` file control the gathering of trace information. After setting the parameters in `SQLNET.ORA`, you must make a new connection for tracing to be performed.

The higher the trace level, the more detail is captured in the trace file. Because the trace file can be hard to understand, start with a trace level of 4 when enabling tracing. The first part of the trace file contains connection handshake information, so look beyond this for the SQL statements and error messages related to your JDBC program.

> **Note:** The trace facility uses a large amount of disk space and might have significant impact upon system performance. Therefore, enable tracing only when necessary.

Client-Side Tracing

Set the following parameters in the `SQLNET.ORA` file on the client system.

> **Note:** Starting from Oracle Database 12*c* Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file `oraaccess.xml` instead of the OCI-specific configuration parameters present in the `sqlnet.ora` file. However, the configuration parameters present in the `sqlnet.ora` file are still supported. For more information about the `oraaccess.xml` file, see *Oracle Call Interface Programmer's Guide*.

TRACE_LEVEL_CLIENT

Purpose:

Turns tracing `on` or `off` to a certain specified level.

Default Value:

0 or OFF

Available Values:

- 0 or `OFF` - No trace output
- 4 or `USER` - User trace information
- 10 or `ADMIN` - Administration trace information
- 16 or `SUPPORT` - WorldWide Customer Support trace information

Example:

```
TRACE_LEVEL_CLIENT=10
```

TRACE_DIRECTORY_CLIENT

Purpose:

Specifies the destination directory of the trace file.

Default Value:

ORACLE_HOME/network/trace

Example:

UNIX: TRACE_DIRECTORY_CLIENT=/oracle/traces

Windows: TRACE_DIRECTORY_CLIENT=C:\ORACLE\TRACES

TRACE_FILE_CLIENT

Purpose:

Specifies the name of the client trace file.

Default Value:

SQLNET.TRC

Example:

TRACE_FILE_CLIENT=cli_Connection1.trc

> **Note:** Ensure that the name you choose for the TRACE_FILE_CLIENT file is different from the name you choose for the TRACE_FILE_SERVER file.

TRACE_UNIQUE_CLIENT

Purpose:

Gives each client-side trace a unique name to prevent each trace file from being overwritten with the next occurrence of a client trace. The PID is attached to the end of the file name.

Default Value:

OFF

Example:

TRACE_UNIQUE_CLIENT = ON

Server-Side Tracing

Set the following parameters in the SQLNET.ORA file on the server system. Each connection will generate a separate file with a unique file name.

> **Note:** Starting from Oracle Database 12*c* Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file oraaccess.xml instead of the OCI-specific configuration parameters present in the sqlnet.ora file. However, the configuration parameters present in the sqlnet.ora file are still supported. For more information about the oraaccess.xml file, see *Oracle Call Interface Programmer's Guide*.

TRACE_LEVEL_SERVER

Purpose:

Turns tracing `on` or `off` to a certain specified level.

Default Value:

0 or `OFF`

Available Values:

- 0 or `OFF` - No trace output
- 4 or `USER` - User trace information
- 10 or `ADMIN` - Administration trace information
- 16 or `SUPPORT` - WorldWide Customer Support trace information

Example:

```
TRACE_LEVEL_SERVER=10
```

TRACE_DIRECTORY_SERVER

Purpose:

Specifies the destination directory of the trace file.

Default Value:

ORACLE_HOME/network/trace

Example:

```
TRACE_DIRECTORY_SERVER=/oracle/traces
```

TRACE_FILE_SERVER

Purpose:

Specifies the name of the server trace file.

Default Value:

```
SERVER.TRC
```

Example:

```
TRACE_FILE_SERVER= svr_Connection1.trc
```

> **Note:** Ensure that the name you choose for the `TRACE_FILE_SERVER` file is different from the name you choose for the `TRACE_FILE_CLIENT` file.

Third Party Debugging Tools

You can use tools such as JDBCSpy and JDBCTest from Intersolv to troubleshoot at the JDBC API level. These tools are similar to ODBC Spy and ODBC Test tools.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

X