Oracle® Database

JDBC Developer's Guide and Reference

10*g* Release 2 (10.2)

B14355-02

June 2006

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

ORACLE®

Oracle Database JDBC Developer's Guide and Reference, 10*g* Release 2 (10.2)

B14355-02

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

Contributing Author: Brian Martin

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

Contents

Preface xix

Audience xix
Documentation Accessibility xix
Related Documents xx
Conventions xxii

Part I Overview

1 Introducing JDBC

Overview of JDBC 1-1
Overview of the Oracle JDBC Drivers 1-1
Common Features of Oracle JDBC Drivers 1-2
JDBC Thin Driver 1-3
JDBC OCI Driver 1-3
JDBC Server-Side Thin Driver 1-3
JDBC Server-Side Internal Driver 1-4
Choosing the Appropriate Driver 1-4
Feature Differences Between JDBC OCI and Thin Drivers 1-5
Overview of Application and Applet Functionality 1-5
Applet Basics 1-5
Oracle Extensions 1-6
Server-Side Basics 1-6
Environments and Support 1-7
Supported JDK and JDBC Versions 1-7
JNI and Java Environments 1-7
JDBC and IDEs 1-7
Changes At This Release 1-7
New Features 1-7
Desupported Features 1-9
Interface Changes 1-9
Feature List 1-10

2 Getting Started

Compatibilities for Oracle JDBC Drivers 2-1
Verifying a JDBC Client Installation 2-1

Checking Installed Directories and Files 2-2
Checking the Environment Variables 2-3
Ensuring that the Java Code Can Be Compiled and Run 2-4
Determining the Version of the JDBC Driver 2-4
Testing JDBC and the Database Connection 2-5

3 Basic Features

Basic Steps in JDBC 3-1
Importing Packages 3-2
Opening a Connection to a Database 3-2
Creating a Statement Object 3-3
Running a Query and Retrieving a ResultSet Object 3-3
Processing the ResultSet Object 3-3
Closing the ResultSet and Statement Objects 3-4
Making Changes to the Database 3-4
Committing Changes 3-5
Closing the Connection 3-5
Sample: Connecting, Querying, and Processing the Results 3-6
Stored Procedure Calls in JDBC Programs 3-6
PL/SQL Stored Procedures 3-6
Java Stored Procedures 3-7
Processing SQL Exceptions 3-7

Part II Oracle JDBC

4 JDBC Standards Support

Introduction 4-1
JDBC 2.0 Support: JDK 1.2.*x* and Later Versions 4-1
Data Type Support 4-2
Standard Feature Support 4-2
Extended Feature Support 4-2
Standard versus Oracle Performance Enhancement APIs 4-2
JDBC 3.0 Support: JDK 1.4 and Previous Releases 4-2
Overview of Supported JDBC 3.0 Features 4-3
Transaction Savepoints 4-3
Creating a Savepoint 4-4
Rolling back to a Savepoint 4-4
Releasing a Savepoint 4-4
Checking Savepoint Support 4-5
Savepoint Notes 4-5
Savepoint Interfaces 4-5
Pre-JDK1.4 Savepoint Support 4-6
Retrieval of Auto-Generated Keys 4-6
java.sql.DatabaseMetaData 4-6
java.sql.Statement 4-6
java.sql.Connection 4-7

Sample Code 4-7
JDBC 3.0 LOB Interface Methods 4-8
Result Set Holdability 4-8

5 Oracle Extensions

Introduction to Oracle Extensions 5-1
Support Features of the Oracle Extensions 5-1
Support for Oracle Data Types 5-2
Support for Oracle Objects 5-2
Support for Schema Naming 5-3
DML Returning 5-4
Accessing PL/SQL Index-by Tables 5-4
Oracle JDBC Packages 5-4
Package oracle.sql 5-4
Package oracle.jdbc 5-10
Oracle Character Data Types Support 5-10
SQL CHAR Data Types 5-10
SQL NCHAR Data Types 5-10
Class oracle.sql.CHAR 5-11
Additional Oracle Type Extensions 5-13
Oracle ROWID Type 5-14
Oracle REF CURSOR Type Category 5-15
Oracle BINARY_FLOAT and BINARY_DOUBLE Types 5-16
The oracle.jdbc Package 5-17
Interface oracle.jdbc.OracleConnection 5-18
Interface oracle.jdbc.OracleStatement 5-19
Interface oracle.jdbc.OraclePreparedStatement 5-20
Interface oracle.jdbc.OracleCallableStatement 5-21
Interface oracle.jdbc.OracleResultSet 5-23
Interface oracle.jdbc.OracleResultSetMetaData 5-24
Class oracle.jdbc.OracleTypes 5-24
Method getJavaSqlConnection 5-25
DML Returning 5-26
Oracle-Specific APIs 5-27
Running DML Returning Statements 5-27
Example of DML Returning 5-28
Limitations of DML Returning 5-29
Accessing PL/SQL Index-by Tables 5-29
Overview 5-29
Binding IN Parameters 5-30
Receiving OUT Parameters 5-31
Type Mappings 5-33

6 Features Specific to JDBC Thin

Introduction to JDBC Thin Client 6-1
Additional Features Supported 6-1

Support for Applets 6-2
Default Support for Native XA 6-2
JDBC in Applets 6-2
Connecting to the Database through the Applet 6-2
Connecting to a Database on a Different Host Than the Web Server 6-3
Using the Oracle Connection Manager 6-3
Using Signed Applets 6-5
Using Applets with Firewalls 6-6
Configuring a Firewall for Applets that use the JDBC Thin Driver 6-6
Writing a URL to Connect through a Firewall 6-7
Packaging Applets 6-8
Specifying an Applet in an HTML Page 6-8
CODE, HEIGHT, and WIDTH 6-8
CODEBASE 6-9
ARCHIVE 6-9

7 Features Specific to JDBC OCI

OCI Connection Pooling 7-1
Transparent Application Failover 7-1
OCI Native XA 7-1
OCI Instant Client 7-1
Overview of Instant Client 7-2
Benefits of Instant Client 7-2
JDBC OCI Instant Client Installation Process 7-3
Usage of Instant Client 7-4
Patching Instant Client Shared Libraries 7-4
Regeneration of Data Shared Library and ZIP files 7-5
Database Connection Names for OCI Instant Client 7-5
Environment Variables for OCI Instant Client 7-7
Instant Client Light (English) 7-8
Globalization Settings 7-9
Operation 7-9
Installation 7-10

8 Server-Side Internal Driver

Introduction 8-1
Connecting to the Database 8-1
Exception-Handling Extensions 8-3
Session and Transaction Context 8-4
Testing JDBC on the Server 8-4
Loading an Application into the Server 8-5
Server-Side Character Set Conversion of oracle.sql.CHAR Data 8-6

Part III Connection and Security

9 Data Sources and URLs

Data Sources 9-1
Overview of Oracle Data Source Support for JNDI 9-1
Features and Properties of Data Sources 9-2
Creating a Data Source Instance and Connecting 9-6
Creating a Data Source Instance, Registering with JNDI, and Connecting 9-6
Supported Connection Properties 9-7
Using Roles for SYS Login 9-9
Configuring Database Remote Login 9-10
Bequeath Connection and SYS Logon 9-11
Properties for Oracle Performance Extensions 9-11
Logging and Tracing 9-12
Database URLs and Database Specifiers 9-13

10 JDBC Client-Side Security Features

Support for Oracle Advanced Security 10-1
JDBC OCI Driver Support for Oracle Advanced Security 10-1
JDBC Thin Driver Support for Oracle Advanced Security 10-2
Support for Login Authentication 10-2
Support for Data Encryption and Integrity 10-2
JDBC OCI Driver Support for Encryption and Integrity 10-3
JDBC Thin Driver Support for Encryption and Integrity 10-4
Setting Encryption and Integrity Parameters in Java 10-5
Secure External Password Store 10-7

11 SSL Support

Overview of SSL 11-1
Java Version of SSL 11-1
SSL in JDBC 11-2
Non-JSSE Related Properties 11-2
JSSE Related Properties 11-4
Enabling SSL 11-5

12 Proxy Authentication

Need for Proxy Authentication 12-1
Creating Proxy Connections 12-2
Caching Proxy Connections 12-3

Part IV Data Access and Manipulation

13 Accessing and Manipulating Oracle Data

Data Type Mappings 13-1
Table of Mappings 13-1
Notes Regarding Mappings 13-3
Data Conversion Considerations 13-3

Standard Types Versus Oracle Types ... 13-4
Converting SQL NULL Data ... 13-4
Testing for NULLs ... 13-4
Result Set and Statement Extensions ... 13-5
Comparison of Oracle get and set Methods to Standard JDBC ... 13-5
Standard getObject Method ... 13-6
Oracle getOracleObject Method ... 13-6
Summary of getObject and getOracleObject Return Types ... 13-7
Other getXXX Methods ... 13-9
Return Types of getXXX Methods ... 13-9
Special Notes about getXXX Methods ... 13-11
Data Types For Returned Objects from getObject and getXXX ... 13-11
The setObject and setOracleObject Methods ... 13-12
Other setXXX Methods ... 13-13
Input Parameter Types of setXXX Methods ... 13-13
Setter Method Size Limitations ... 13-15
Setter Methods That Take Additional Input ... 13-15
Method setFixedCHAR for Binding CHAR Data into WHERE Clauses ... 13-16
Using Result Set Meta Data Extensions ... 13-17

14 Java Streams in JDBC

Overview ... 14-1
Streaming LONG or LONG RAW Columns ... 14-2
LONG RAW Data Conversions ... 14-2
LONG Data Conversions ... 14-2
Streaming Example for LONG RAW Data ... 14-3
Avoiding Streaming for LONG or LONG RAW ... 14-5
Streaming CHAR, VARCHAR, or RAW Columns ... 14-5
Streaming LOBs and External Files ... 14-6
Data Streaming and Multiple Columns ... 14-7
Closing a Stream ... 14-8
Notes and Precautions on Streams ... 14-8
Streaming Data Precautions ... 14-8
Using Streams to Avoid Limits on setBytes and setString ... 14-9
Streaming and Row Prefetching ... 14-9

15 Working with Oracle Object Types

Mapping Oracle Objects ... 15-1
Using the Default STRUCT Class for Oracle Objects ... 15-2
STRUCT Class Functionality ... 15-2
Retrieving STRUCT Objects and Attributes ... 15-4
Creating STRUCT Objects and Descriptors ... 15-5
Binding STRUCT Objects into Statements ... 15-7
STRUCT Automatic Attribute Buffering ... 15-7
Creating and Using Custom Object Classes for Oracle Objects ... 15-8
Relative Advantages of ORAData versus SQLData ... 15-9
Understanding Type Maps for SQLData Implementations ... 15-9

Creating Type Map and Defining Mappings for a SQLData Implementation 15-10
Adding Entries to an Existing Type Map 15-10
Creating a New Type Map 15-11
Materializing Object Types not Specified in the Type Map 15-11
Understanding the SQLData Interface 15-12
Reading and Writing Data with a SQLData Implementation 15-13
Understanding the ORAData Interface 15-16
Reading and Writing Data with a ORAData Implementation 15-17
Additional Uses for ORAData 15-20
The Deprecated CustomDatum Interface 15-20
Object-Type Inheritance 15-21
Creating Subtypes 15-21
Implementing Customized Classes for Subtypes 15-22
Use of ORAData for Type Inheritance Hierarchy 15-22
Use of SQLData for Type Inheritance Hierarchy 15-25
JPublisher Utility 15-27
Retrieving Subtype Objects 15-27
Creating Subtype Objects 15-29
Sending Subtype Objects 15-30
Accessing Subtype Data Fields 15-30
Inheritance Meta Data Methods 15-31
Using JPublisher to Create Custom Object Classes 15-32
JPublisher Functionality 15-32
JPublisher Type Mappings 15-33
Describing an Object Type 15-35
Functionality for Getting Object Meta Data 15-35
Steps for Retrieving Object Meta Data 15-36

16 Working with LOBs and BFILEs

Oracle Extensions for LOBs and BFILEs 16-1
Working with BLOBs and CLOBs 16-2
Getting and Passing BLOB and CLOB Locators 16-2
Retrieving BLOB and CLOB Locators 16-2
Passing BLOB and CLOB Locators 16-3
Reading and Writing BLOB and CLOB Data 16-4
Creating and Populating a BLOB or CLOB Column 16-7
Accessing and Manipulating BLOB and CLOB Data 16-8
Additional BLOB and CLOB Features 16-9
Data Interface for LOBs 16-12
Working With Temporary LOBs 16-14
Using Open and Close With LOBs 16-16
Working with BFILEs 16-16
Getting and Passing BFILE Locators 16-16
Reading BFILE Data 16-18
Creating and Populating a BFILE Column 16-19
Accessing and Manipulating BFILE Data 16-20
Additional BFILE Features 16-21

17 Using Oracle Object References

Oracle Extensions for Object References ... 17-1
Overview of Object Reference Functionality ... 17-2
Object Reference Getter and Setter Methods ... 17-2
Key REF Class Methods ... 17-2
Retrieving and Passing an Object Reference ... 17-3
Retrieving an Object Reference from a Result Set ... 17-3
Retrieving an Object Reference from a Callable Statement ... 17-4
Passing an Object Reference to a Prepared Statement ... 17-4
Accessing and Updating Object Values through an Object Reference ... 17-5
Custom Reference Classes with JPublisher ... 17-5

18 Working with Oracle Collections

Oracle Extensions for Collections ... 18-1
Choices in Materializing Collections ... 18-1
Creating Collections ... 18-2
Creating Multilevel Collection Types ... 18-3
Overview of Collection Functionality ... 18-3
Array Getter and Setter Methods ... 18-3
ARRAY Descriptors and ARRAY Class Functionality ... 18-4
ARRAY Performance Extension Methods ... 18-5
Accessing oracle.sql.ARRAY Elements as Arrays of Java Primitive Types ... 18-5
ARRAY Automatic Element Buffering ... 18-6
ARRAY Automatic Indexing ... 18-6
Creating and Using Arrays ... 18-7
Creating ARRAY Objects and Descriptors ... 18-7
Retrieving an Array and Its Elements ... 18-10
Retrieving the Array ... 18-10
Data Retrieval Methods ... 18-11
Comparing the Data Retrieval Methods ... 18-12
Retrieving Elements of a Structured Object Array According to a Type Map ... 18-12
Retrieving a Subset of Array Elements ... 18-13
Retrieving Array Elements into an oracle.sql.Datum Array ... 18-13
Accessing Multilevel Collection Elements ... 18-14
Passing Arrays to Statement Objects ... 18-15
Using a Type Map to Map Array Elements ... 18-16
Custom Collection Classes with JPublisher ... 18-18

19 Result Set

Overview ... 19-1
Result Set Functionality and Result Set Categories Supported in JDBC 2.0 ... 19-1
Oracle JDBC Implementation Overview for Result Set Enhancements ... 19-3
Creating Scrollable or Updatable Result Sets ... 19-5
Specifying Result Set Scrollability and Updatability ... 19-5
Result Set Limitations and Downgrade Rules ... 19-6
Positioning and Processing in Scrollable Result Sets ... 19-8

Positioning in a Scrollable Result Set 19-8
Processing a Scrollable Result Set 19-10
Updating Result Sets 19-11
Performing a DELETE Operation in a Result Set 19-11
Performing an UPDATE Operation in a Result Set 19-12
Performing an INSERT Operation in a Result Set 19-13
Avoiding Update Conflicts 19-14
Fetch Size 19-15
Setting the Fetch Size 19-15
Use of Standard Fetch Size versus Oracle Row-Prefetch Setting 19-16
Refetching Rows 19-16
Seeing Database Changes Made Internally and Externally 19-17
Seeing Internal Changes 19-17
Seeing External Changes 19-18
Visibility versus Detection of External Changes 19-18
Summary of Visibility of Internal and External Changes 19-19
Oracle Implementation of Scroll-Sensitive Result Sets 19-19
Summary of New Methods for Result Set Enhancements 19-20
Modified Connection Methods 19-20
New Result Set Methods 19-20
Statement Methods 19-22
Database Meta Data Methods 19-22

20 JDBC RowSets

Overview 20-1
RowSet Properties 20-2
Events and Event Listeners 20-2
Command Parameters and Command Execution 20-4
Traversing RowSets 20-4
CachedRowSet 20-6
JDBCRowSet 20-8
WebRowSet 20-9
FilteredRowSet 20-11
JoinRowSet 20-13

21 Globalization Support

Providing Globalization Support 21-1
NCHAR, NVARCHAR2, NCLOB and the defaultNChar Property 21-2

Part V Performance Enhancements

22 Statement Caching

About Statement Caching 22-1
Basics of Statement Caching 22-1
Implicit Statement Caching 22-2
Explicit Statement Caching 22-2

Using Statement Caching 22-3
Enabling and Disabling Statement Caching 22-4
Physically Closing a Cached Statement 22-5
Using Implicit Statement Caching 22-5
Using Explicit Statement Caching 22-6

23 Implicit Connection Caching

The Implicit Connection Cache 23-2
Using the Connection Cache 23-3
Turning Caching On 23-3
Opening a Connection 23-4
Setting Connection Cache Name 23-4
Setting Connection Cache Properties 23-5
Closing A Connection 23-5
Implicit Connection Cache Example 23-5
Connection Attributes 23-6
Getting Connections 23-7
Setting Connection Attributes 23-7
Checking Attributes of a Returned Connection 23-8
Connection Attribute Example 23-8
Connection Cache Properties 23-8
Limit Properties 23-9
TIMEOUT Properties 23-10
Other Properties 23-10
Connection Property Example 23-11
Connection Cache Manager API 23-12
createCache 23-12
disableCache 23-12
enableCache 23-13
existsCache 23-13
getCacheNameList 23-13
getCacheProperties 23-13
getNumberOfActiveConnections 23-13
getNumberOfAvailableConnections 23-13
isFatalConnectionError 23-14
purgeCache 23-14
refreshCache 23-14
reinitializeCache 23-14
removeCache 23-15
setConnectionPoolDataSource 23-15
Example Of ConnectionCacheManager Use 23-15
Advanced Topics 23-16
Attribute Weights And Connection Matching 23-16
Connection Cache Callbacks 23-17
Use Cases for TimeToLiveTimeout and AbandonedConnectionTimeout 23-17

24 Run-Time Connection Load Balancing

Overview 24-1
Run-Time Connection Load Balancing 24-1
Enabling Run-Time Connection Load Balancing 24-2

25 Performance Extensions

Update Batching 25-1
Overview of Update Batching Models 25-2
Oracle Update Batching 25-3
Standard Update Batching 25-8
Premature Batch Flush 25-13
Additional Oracle Performance Extensions 25-14
Oracle Row Prefetching 25-15
Setting the Oracle Prefetch Value 25-15
Oracle Row-Prefetching Limitations 25-17
Defining Column Types 25-17
DatabaseMetaData TABLE_REMARKS Reporting 25-20

26 OCI Connection Pooling

OCI Driver Connection Pooling: Background 26-1
OCI Driver Connection Pooling and Shared Servers Compared 26-2
Defining an OCI Connection Pool 26-2
Connecting to an OCI Connection Pool 26-6
Sample Code for OCI Connection Pooling 26-7
Statement Handling and Caching 26-9
JNDI and the OCI Connection Pool 26-10

Part VI High Availability

27 Fast Connection Failover

Introduction 27-1
Using Fast Connection Failover 27-2
Fast Connection Failover Prerequisites 27-2
Configuring ONS For Fast Connection Failover 27-2
ONS Configuration File 27-2
Client-Side ONS Configuration 27-4
Server-Side ONS Configuration Using racgons 27-4
Remote ONS Subscription 27-5
Enabling Fast Connection Failover 27-6
Querying Fast Connection Failover Status 27-7
Understanding Fast Connection Failover 27-7
What The Application Sees 27-7
How It Works 27-7
Comparison of Fast Connection Failover and TAF 27-8

28 Transparent Application Failover

Overview 28-1
Failover Type Events 28-1
TAF Callbacks 28-2
Java TAF Callback Interface 28-2

Part VII Transaction Management

29 Distributed Transactions

Overview 29-1
Distributed Transaction Components and Scenarios 29-2
Distributed Transaction Concepts 29-2
Switching Between Global and Local Transactions 29-4
Oracle XA Packages 29-5
XA Components 29-6
XADatasource Interface and Oracle Implementation 29-6
XAConnection Interface and Oracle Implementation 29-6
XAResource Interface and Oracle Implementation 29-7
OracleXAResource Method Functionality and Input Parameters 29-8
Xid Interface and Oracle Implementation 29-13
Error Handling and Optimizations 29-14
XAException Classes and Methods 29-14
Mapping between Oracle Errors and XA Errors 29-14
XA Error Handling 29-15
Oracle XA Optimizations 29-15
Implementing a Distributed Transaction 29-15
Summary of Imports for Oracle XA 29-15
Oracle XA Code Sample 29-16
Native-XA in Oracle JDBC Drivers 29-20
OCI Native XA 29-20
Thin Native XA 29-21

Part VIII Manageability

30 End-To-End Metrics Support

Introduction 30-1
JDBC API For End-To-End Metrics 30-2

Part IX Appendixes

A Reference Information

Valid SQL-JDBC Data Type Mappings A-1
Supported SQL and PL/SQL Data Types A-3
Embedded SQL92 Syntax A-6
Time and Date Literals A-7

Scalar Functions ... A-8
LIKE Escape Characters ... A-9
Outer Joins ... A-9
Function Call Syntax ... A-9
SQL92 to SQL Syntax Example ... A-10
Oracle JDBC Notes and Limitations ... A-10
CursorName ... A-11
SQL92 Outer Join Escapes ... A-11
PL/SQL TABLE, BOOLEAN, and RECORD Types ... A-11
IEEE 754 Floating Point Compliance ... A-11
Catalog Arguments to DatabaseMetaData Calls ... A-11
SQLWarning Class ... A-12
Binding Named Parameters ... A-12

B Coding Tips

JDBC and Multithreading ... B-1
Performance Optimization ... B-1
Disabling Auto-Commit Mode ... B-1
Standard Fetch Size and Oracle Row Prefetching ... B-2
Standard and Oracle Update Batching ... B-2
Statement Caching ... B-3
Mapping Between Built-in SQL and Java Types ... B-3
Transaction Isolation Levels and Access Modes ... B-4

C JDBC Error Messages

General Structure of JDBC Error Messages ... C-1
General JDBC Messages ... C-1
JDBC Messages Sorted by ORA Number ... C-1
JDBC Messages Sorted in Alphabetic Order ... C-5
Native XA Messages ... C-9
Native XA Messages Sorted by ORA Number ... C-9
Native XA Messages Sorted in Alphabetic Order ... C-9
TTC Messages ... C-10
TTC Messages Sorted by ORA Number ... C-10
TTC Messages Sorted in Alphabetic Order ... C-11

D Troubleshooting

Common Problems ... D-1
Memory Consumption for CHAR Columns Defined as OUT or IN/OUT Variables ... D-1
Memory Leaks and Running Out of Cursors ... D-2
Boolean Parameters in PL/SQL Stored Procedures ... D-2
Opening More Than 16 OCI Connections for a Process ... D-2
Using statement.cancel ... D-3
Using JDBC with Firewalls ... D-3
Basic Debugging Procedures ... D-4
Oracle Net Tracing to Trap Network Events ... D-4

Client-Side Tracing D-4
TRACE_LEVEL_CLIENT D-4
TRACE_DIRECTORY_CLIENT D-5
TRACE_FILE_CLIENT D-5
TRACE_UNIQUE_CLIENT D-5
Server-Side Tracing D-6
TRACE_LEVEL_SERVER D-6
TRACE_DIRECTORY_SERVER D-6
TRACE_FILE_SERVER D-6
Third Party Debugging Tools D-6

Index

List of Tables

1–1	Feature Differences Between JDBC OCI and JDBC Thin Drivers	1-5
1–2	Feature List	1-10
3–1	Import Statements for JDBC Driver	3-2
4–1	JDBC 3.0 Feature Support	4-3
4–2	Key Areas of JDBC 3.0 Functionality	4-3
4–3	BLOB Method Equivalents	4-8
4–4	CLOB Method Equivalents	4-8
5–1	Oracle Data Type Classes	5-5
5–2	Key Interfaces and Classes of the oracle.jdbc Package	5-17
5–3	PL/SQL Types and Corresponding JDBC Types	5-29
5–4	Arguments of the setPlsqlIndexTable Method	5-30
5–5	Arguments of the registerIndexTableOutParameter Method	5-32
5–6	Argument of the getPlsqlIndexTable Method	5-33
5–7	Argument of the getOraclePlsqlIndexTable Method	5-34
5–8	Arguments of the getPlsqlIndexTable Method	5-34
7–1	OCI Instant Client Shared Libraries	7-2
7–2	Data Shared Library for Instant Client and Instant Client Light (English)	7-8
9–1	Standard Data Source Properties	9-3
9–2	Oracle Extended Data Source Properties	9-4
9–3	Connection Properties Recognized by Oracle JDBC Drivers	9-8
9–4	Supported Database Specifiers	9-14
10–1	Client/Server Negotiations for Encryption or Integrity	10-3
10–2	OCI Driver Client Parameters for Encryption and Integrity	10-4
10–3	Thin Driver Client Parameters for Encryption and Integrity	10-4
13–1	Default Mappings Between SQL Types and Java Types	13-2
13–2	getObject and getOracleObject Return Types	13-7
13–3	Summary of getXXX Return Types	13-10
13–4	Summary of setXXX Input Parameter Types	13-13
13–5	Size Limitations for setBytes and setString Methods	13-15
14–1	LONG and LONG RAW Data Conversions	14-3
15–1	JPublisher SQL Type Categories, Supported Settings, and Defaults	15-35
19–1	Visibility of Internal and External Changes for Oracle JDBC	19-19
20–1	The JDBC and Cached Row Sets Compared	20-9
22–1	Comparing Methods Used in Statement Caching	22-3
22–2	Methods Used in Statement Allocation and Implicit Statement Caching	22-6
22–3	Methods Used to Retrieve Explicitly Cached Statements	22-8
25–1	Valid Column Type Specifications	25-19
27–1	onsctl commands	27-4
29–1	Connection Mode Transitions	29-4
29–2	Oracle-XA Error Mapping	29-14
30–1	Maximum Lengths for End-to-End Metrics	30-1
A–1	Valid SQL Data Type-Java Class Mappings	A-1
A–2	Support for SQL Data Types	A-3
A–3	Support for ANSI-92 SQL Data Types	A-4
A–4	Support for SQL User-Defined Types	A-4
A–5	Support for PL/SQL Data Types	A-5
B–1	Mapping of SQL Data Types to Java Classes that Represent SQL Data Types	B-3

Preface

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

Audience

The *Oracle Database JDBC Developer's Guide and Reference* 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

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

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

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

TTY Access to Oracle Support Services

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

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 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 Application Server 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 Application Server 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 Application Server 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 Application Server 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 Application Server 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 Application Developer's Guide - Fundamentals*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database PL/SQL User's Guide and Reference*
- *Oracle Database SQL Reference*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database Advanced Security Administrator's 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 10g Administrator's Guide*
- *Oracle Enterprise Manager Administrator's Guide*
- *Oracle HTTP Server Administrator's Guide*
- *Oracle Application Server 10g Performance Guide*
- *Oracle Application Server 10g Globalization Guide*
- *Oracle Application Server Web Cache Administrator's Guide*
- *Oracle Application Server 10g Upgrading to 10g (9.0.4)*

The following are available from the JDeveloper group:

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

  `http://otn.oracle.com/products/jdev/content.html`

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

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

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

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

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

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

The following resources are available from Sun Microsystems:

- Web site for JDBC, including the latest specifications:

  `http://java.sun.com/products/jdbc/`

- `jdbc-interest` discussion group for JDBC

  To subscribe, send an e-mail to `listserv@java.sun.com` with the following line in the body of the message:

  ```
  subscribe jdbc-interest yourlastname yourfirstname
  ```

  We recommend that you request only the daily digest of the posted e-mails. To do this add the following line to the message body as well:

```
set jdbc-interest digest
```

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]`

`...`	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;`

Convention	Meaning	Example

`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.
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 scott/tiger TABLES=emp QUERY=\"WHERE job='SALESMAN' and sal<1600\"` `C:\>imp SYSTEM/`*`password`*` FROMUSER=scott TABLES=(emp, 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 (32-Bit)* 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.

Part I

Overview

This part consists of chapters that introduce the concept of Java Database Connectivity (JDBC) and provide an overview of the Oracle implementation of JDBC. It provides basic information on installation and configuration of Oracle client with reference to JDBC drivers. It also includes chapters that cover the basic steps in creating and running any JDBC application.

Part I contains the following chapters:

- Chapter 1, "Introducing JDBC"
- Chapter 2, "Getting Started"
- Chapter 3, "Basic Features"

1

Introducing JDBC

This chapter provides an overview of the Oracle implementation of Java Database Connectivity (JDBC), covering the following topics:

- Overview of JDBC
- Overview of the Oracle JDBC Drivers
- Overview of Application and Applet Functionality
- Server-Side Basics
- Environments and Support
- Changes At This Release

Overview of JDBC

JDBC is a Java standard that provides the interface for connecting from Java to relational databases. The JDBC standard is defined by Sun Microsystems and implemented through the standard `java.sql` interfaces. This allows 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) and complies with the SQL92 Entry Level standard.

Overview of the 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

  It is a pure Java driver used on the client side, without an Oracle client installation. It can be used with both applets and applications.

- Oracle Call Interface (OCI) driver

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

- Server-side Thin driver

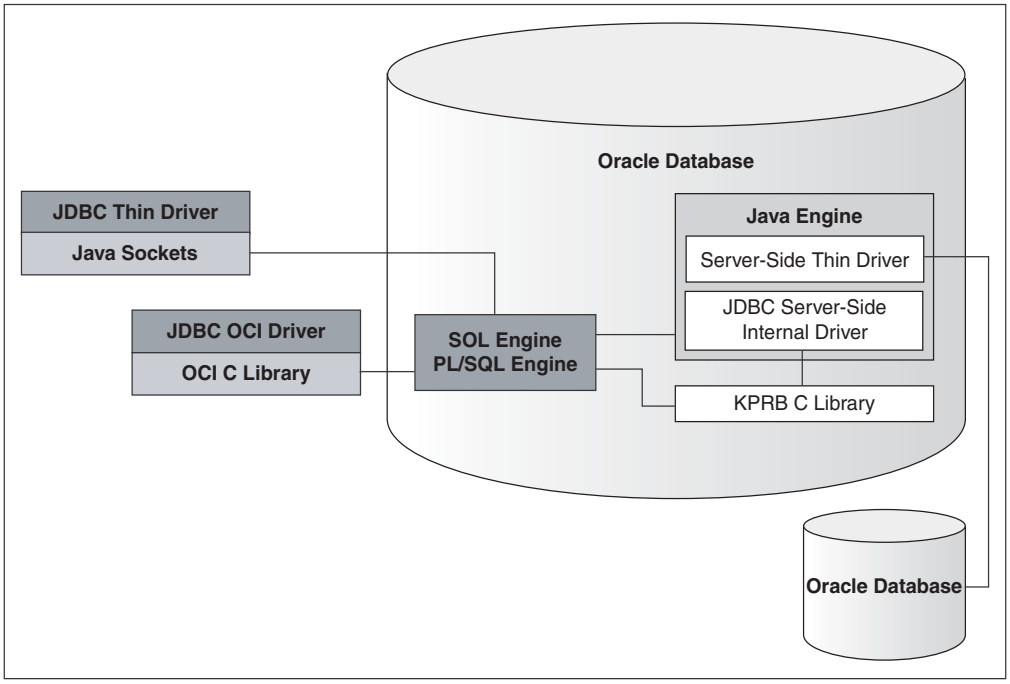
  It is functionally similar to the client-side Thin driver. However, it is used for code that runs on the database server and needs to access another session either on the same server or on a remote server on any tier.

- 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.

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

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



This section covers the following topics:

- Common Features of Oracle JDBC Drivers
- JDBC Thin Driver
- JDBC OCI Driver
- JDBC Server-Side Thin Driver
- JDBC Server-Side Internal Driver
- Choosing the Appropriate Driver

Common Features of Oracle JDBC Drivers

The server-side and client-side Oracle JDBC drivers provide the same basic functionality.

The JDBC Thin and OCI drivers support the following versions of Java Development Kit (JDK): 1.2.x, 1.3.x and 1.4.x. The server-side Thin driver and server-side internal driver support JDK 1.4.1. All the JDBC drivers support the following standards and features:

- Same syntax and APIs
- Same Oracle extensions
- Full support for multithreaded applications

Oracle JDBC drivers implement the standard `java.sql` interfaces. You can access the Oracle-specific features, in addition to the standard features, by using the `oracle.jdbc` package.

JDBC 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 SQL*Net to access the Oracle Database.

The JDBC Thin driver allows a direct connection to the database by providing an implementation of SQL*Net 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.

See Also: Chapter 6, "Features Specific to JDBC Thin"

JDBC OCI Driver

The JDBC OCI driver is a Type II driver used with Java applications. It requires an Oracle client installation and, therefore, is Oracle platform-specific. It supports all installed Oracle Net adapters, including 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 SQL*Net.

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 7, "Features Specific to JDBC OCI"

JDBC Server-Side Thin Driver

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 driven runs inside the Oracle Database and accesses a remote database or a different session on the same 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 6, "Features Specific to JDBC Thin"

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 `SCOTT`:

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

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*

JDBC Server-Side Internal Driver

The JDBC server-side internal driver supports any Java code that runs inside the Oracle Database, such as in a Java stored procedures or Enterprise JavaBean (EJB), and must access the same database. It lets the Java virtual machine (JVM) to communicate directly with the SQL engine. The driver supports only JDK 1.4.1.

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 8, "Server-Side Internal Driver"

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 the 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 acting as a middle tier, use the JDBC server-side Thin driver.
- If your code will run inside the target database server, 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 10*g* release 2 (10.2).

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

JDBC OCI Driver	JDBC Thin Driver
OCI connection pooling	Support for applets
OCI optimized fetch	Default support for Native XA
Client-side object cache	
Transparent Application Failover (TAF)	
OCI Instant Client	
Instant Client Light (English)	
Strong authentication; Kerberos, PKI certificates	

Notes:

- 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.
- Most JDBC OCI driver features are not available in the JDBC Thin driver because they are inherited from OCI.

See Also:

- Chapter 6, "Features Specific to JDBC Thin"
- Chapter 7, "Features Specific to JDBC OCI"

Overview of Application and Applet Functionality

This section compares and contrasts the basic functionality of JDBC applications and applets. It also introduces the Oracle extensions that can be used by application and applet programmers. This sections covers the following topics:

- Applet Basics
- Oracle Extensions

Applet Basics

You can use only the Oracle JDBC Thin driver for an applet.

See Also: "JDBC in Applets" on page 6-2

Applets and Security

An applet can open network connections only to the host computer from which it was downloaded. Therefore, an applet can connect only to databases on the originating computer. 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 take advantage of the data encryption and integrity checksum features of the Oracle Advanced Security option.

Applets and Firewalls

An applet can connect to a database through a firewall.

See Also: "Using Applets with Firewalls" on page 6-6

Packaging and Deploying Applets

To package and deploy an applet, you must place the JDBC Thin driver classes and the applet classes in the same `.zip` file.

See Also: "Packaging Applets" on page 6-8

Oracle Extensions

A number of Oracle extensions are available, which can be used by Oracle JDBC application and applet programmers. These extensions include:

- Type extensions, such as `ROWID` and `REF CURSOR` types
- Wrapper classes for SQL types provided by the `oracle.sql` package
- Support for custom Java classes to map to user-defined types
- Extended large object (LOB) support
- Extended connection, statement, and result set functionality
- Performance enhancements

See Also: Chapter 5, "Oracle Extensions" and Chapter 25, "Performance Extensions"

Server-Side Basics

By using the JDBC server-side internal driver, code that runs in the Oracle Database, such as Java stored procedures or EJBs, can access the database in which it runs.

See Also: Chapter 8, "Server-Side Internal Driver"

Session and Transaction Context

The JDBC server-side internal driver operates within a default session and default transaction context.

See Also: "Session and Transaction Context" on page 8-4.

Connecting to the Database

The JDBC server-side internal driver uses a default connection to the database. You connect to the database with the `OracleDataSource.getconnection` method.

See Also: "Connecting to the Database" on page 8-1.

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 10*g* release 2 (10.2), all the JDBC drivers are compatible with JDK 1.2.x and later. JDK 1.0.x and JDK 1.1.x are no longer supported, and therefore, the `classes111.zip`, `classes111.jar`, `classes111_g.zip`, `classes111_g.jar`, and `nls_charset11.zip` files are no longer provided.

See Also: "Compatibilities 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 JVMs other than that of Sun Microsystems, 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.

Changes At This Release

The Oracle implementation JDBC provides many enhancements in Oracle Database 10*g*. This section gives an overview of these enhancements. It is divided into the following sections:

- New Features
- Desupported Features
- Interface Changes

New Features

Oracle Database 10*g* release 2 (10.2) supports the following new features:

- DML returning

The JDBC OCI and the JDBC Thin drivers support the data manipulation language (DML) returning feature. DML returning enables you to retrieve auto-generated keys along with other columns or values that your application may use.

See Also: "DML Returning" on page 5-4

- JSR 114 RowSets

  Oracle Database 10*g* release 2 (10.2) provides support for all the RowSet implementations defined in the JDBC RowSet Implementations Specification (JSR-114). In particular, Oracle Database 10*g* release 2 (10.2) provides support for WebRowSet, FilteredRowSet, and JoinRowSet.

  See Also: Chapter 20, "JDBC RowSets"

- `NCHAR` literal support

  Oracle Database 10*g* release 2 (10.2) provides the NCHAR literal support. For further information, refer to the Note in "NCHAR, NVARCHAR2, NCLOB and the defaultNChar Property" on page 21-2.

- Proxy authentication

  In this release of Oracle Database, the proxy authentication feature is supported by both the JDBC Thin and JDBC OCI drivers.

  See Also: Chapter 12, "Proxy Authentication"

- Result set holdability

  Oracle JDBC drivers provide support for result set holdability, which is a feature of the JDBC 3.0 standard. This feature enables applications to decide whether the `ResultSet` objects should be open or closed, when an implicit or explicit commit operation is performed.

  See Also: "Result Set Holdability" on page 4-8

- Retrieval of auto-generated keys

  Oracle Database 10*g* release 2 (10.2) provides support for the retrieval of auto-generated keys JDBC 3.0 standard feature. This feature enables you to retrieve values that are generated by the database.

  See Also: "Retrieval of Auto-Generated Keys" on page 4-6

- Run-time connection load balancing

  The Oracle JDBC Thin and JDBC OCI drivers support the run-time connection load balancing feature. This feature enables routing of work requests to a database instance that offers the best performance, minimizing the need to relocate work.

  See Also: Chapter 24, "Run-Time Connection Load Balancing"

- SSL support

  Oracle Database 10*g* release 2 (10.2) provides support for the Secure Sockets Layer (SSL) Protocol.

  See Also: Chapter 11, "SSL Support"

- XAConnection caching

  The implicit connection caching feature supports the caching of XA connections. This feature is supported by both the JDBC Thin and JDBC OCI drivers.

  See Also: Chapter 23, "Implicit Connection Caching"

Desupported Features

No feature has been desupported in Oracle Database 10*g* release 2 (10.2). That is, Oracle Database 10*g* release 2 (10.2) supports all features that were supported in Oracle Database 10*g* release 1 (10.1).

Interface Changes

There are some changes in the `setString`, `setCharacterStream`, `setAsciiStream`, `setBytes`, and `setBinaryStream` methods of `PreparedStatement` in Oracle Database 10*g* release 2 (10.2).

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 normal 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 32512.

The original behavior of the 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

In Oracle Database 10*g* release 2 (10.2), 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 32512 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 32512 and LOB bind 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 32512. For variable length character sets, there are three cases based on the Java character length, as follows:

- If character length is less than 32512 divided by the maximum character size, then direct bind is used.
- If character length is greater than 32512 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 32512, 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 16-12 for more information.

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		8.0.6	8.0.6
Row Prefetch		8.0.6	8.0.6
Oracle Batching		8.0.6	8.0.6
JNI 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
Index-By Tables	10.2.0	8.1.6	10.1.0
JDBC 2.0 Scrollable ResultSets	8.1.6	8.1.6	8.1.6
JDBC 2.0 Updatable ResultSets	8.1.6	8.1.6	8.1.6
JDBC 2.0 Standard Batching	8.1.6	8.1.6	8.1.6
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

Table 1–2 (Cont.) Feature List

Feature	Server-Side Internal	JDBC OCI	JDBC Thin
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 ResultSets	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		10.1.0	10.1.0
JDK 5.0		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
JDBC 3.0 Local/Global Transaction Switching	9.2.0	9.2.0	9.2.0

Table 1–2 (Cont.) Feature List

Feature	Server-Side Internal	JDBC OCI	JDBC Thin
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	
Radius Authentication		10.2.0	

Notes:

- In the table, NA implies that the feature is not applicable for the corresponding Oracle JDBC driver.
- The `ConnectionCacheImpl` connection cache feature is deprecated in Oracle Database 10*g*. Implicit Connection Cache replaces this in Oracle Database 10*g*.

2

Getting Started

This chapter provides a discussion on the compatibilities between 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:

- Compatibilities for Oracle JDBC Drivers
- Verifying a JDBC Client Installation

Compatibilities for Oracle JDBC Drivers

This section discusses the general JDBC version compatibility issues.

Backward Compatibility

The JDBC drivers are certified to work with the currently supported versions of the Oracle Database. For example, the JDBC Thin drivers in Oracle Database 10*g* are certified to work with the 9.2.x, 9.0.1.x, and 8.1.7 Oracle Database releases. However, they are not certified to work with older, unsupported database releases, such as 8.0.x and 7.x.

Forward Compatibility

Existing and supported JDBC drivers are certified to work with Oracle Database 10*g*.

Notes:

- In Oracle Database 10*g*, the Oracle JDBC drivers no longer support JDK 1.1.x or earlier versions.
- You can find a complete, up-to-date list of supported databases at `http://www.oracle.com/technology/tech/java/sqlj_jdbc/htdocs/jdbc_faq.htm`.

Verifying a JDBC Client Installation

Verifying a JDBC client installation involves:

- Checking Installed Directories and Files
- Checking the Environment Variables
- Ensuring that the Java Code Can Be Compiled and Run

- Determining the Version of the JDBC Driver
- Testing JDBC and the Database Connection

Installation of an Oracle JDBC driver is platform-specific. Follow the installation instructions for the driver you want to install in your platform-specific documentation.

This section describes the steps of verifying an Oracle client installation of the JDBC drivers, assuming that you have already installed the driver of your choice.

If you have installed the JDBC Thin driver, then no further installation on the client computer is necessary.

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

If you have installed the JDBC Oracle Call Interface (OCI) driver, then you must also install the Oracle client software. This includes Oracle Net and the OCI libraries.

Checking Installed Directories and Files

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

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

Check whether the following directories 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 is created. The samples directory contains sample programs, including examples of how to use SQL92 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

    Contains classes for globalization and multibyte character sets support

  - classes12.jar and classes12_g.jar

    Contain the JDBC driver classes for use with JDK releases after 1.2 and before 1.4

  - ojdbc14.jar and ojdbc14_g.jar

    Contain the JDBC driver classes for use with JDK 1.4

- 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 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) for JDK 1.2.x, 1.3.x, and 1.4. These are only required if you are using JTA features for distributed transaction management or JNDI features for naming services.

> **Note:** These files can also be obtained from the Sun Microsystems Web site. However, it is recommend to use the versions supplied by Oracle, which have been tested with the Oracle drivers.

Checking 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 the Sun Solaris and Microsoft Windows platforms.

You must set the CLASSPATH for your installed JDBC OCI or Thin driver. Depending on which JDK version you use, you must set one of these values for the CLASSPATH:

JDK Version	CLASSPATH
1.4	*ORACLE_HOME*/jdbc/lib/ojdbc14.jar *ORACLE_HOME*/jlib/orai18n.jar
1.3.x, 1.2.x	*ORACLE_HOME*/jdbc/lib/classes12.jar *ORACLE_HOME*/jlib/orai18n.jar

Ensure that there is only one JDBC class file, such as classes12.jar, classes12_g.jar, or ojdbc14.jar, and one globalization classes file, orai18n.jar, in your CLASSPATH.

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

JDBC OCI Driver

If you are installing the JDBC OCI driver, then you must also set the following value for the library path environment variable:

- On Sun Solaris, set LD_LIBRARY_PATH as follows:

  ORACLE_HOME/lib

  This directory contains the libocijdbc10.so shared object library.

> **Note:** If you are running a 32-bit Java virtual machine (JVM) against a 64-bit client or database, then you must also add *ORACLE_HOME*/lib32 to LD_LIBRARY_PATH.

- On Microsoft Windows, set `PATH` as follows:

  ORACLE_HOME\bin

  This directory contains the `ocijdbc10.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 7, "Features Specific to JDBC OCI"

JDBC Thin Driver

If you are installing the JDBC Thin driver, then you do not have to set any other environment variables.

Ensuring that the Java Code Can Be Compiled and Run

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

```
javac

java
```

Each of the preceding commands should output 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`.

Determining the Version of the JDBC Driver

You can determine the version of the JDBC driver that you installed, by calling the `getDriverVersion` method of the `OracleDatabaseMetaData` class.

Following is a sample code illustrating how to determine the driver version:

```
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:scott/tiger@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());
  }
}
```

Testing 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.

Traverse to the `samples` directory, and compile and run `JdbcCheckup.java`. 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
- Outputs 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 connect
    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 "";
    }
  }
}
```

3

Basic Features

This chapter covers the most basic steps that are taken in creating any Java Database Connectivity (JDBC) application. It also describes the basic features of Java and JDBC supported by the Oracle JDBC drivers.

The following topics are discussed:

- Basic Steps in JDBC
- Sample: Connecting, Querying, and Processing the Results
- Stored Procedure Calls in JDBC Programs
- Processing SQL Exceptions

Basic Steps in JDBC

When using the 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 creating code to connect to and query 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 ResultSet Object
5. Processing the ResultSet Object
6. Closing the ResultSet 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, which allow 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 3–1 at the beginning of your program.

Table 3–1 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 the 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 document uses asterisk (`*`) for simplicity, but this is not the recommend 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 `OracleDataSource.getConnection`. 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.

> **See Also:** Table 9–3, " Connection Properties Recognized by Oracle JDBC Drivers"

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 `scott` with password `tiger` to a database with service `orcl` through port 1521 of the host `myhost`, using the JDBC Thin driver.

```
OracleDataSource ods = new OracleDataSource();
String url = "jdbc:oracle:thin:@//myhost:1521/orcl",
ods.setURL(url);
ods.setUser("scott");
ods.setPassword("tiger");
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 `scott` with password `tiger` to a database host whose 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:scott/tiger@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 `myhost` that has a TCP/IP listener on port 1521 and the service identifier is `orcl`, then provide the following code:

```
String URL = "jdbc:oracle:thin:scott/tiger@//myhost:1521/orcl");
ods.setURL(URL);
Connection conn = ods.getConnection();
```

See Also: Chapter 9, "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 ResultSet 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.

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 `ENAME` column of a table of employees named `EMP`:

```
ResultSet rset = stmt.executeQuery ("SELECT ename FROM emp");
```

Processing the ResultSet 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 ResultSet 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 the 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

To write changes to the database, such as for `INSERT` or `UPDATE` operations, you create a `PreparedStatement` object. This enables 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 13-12 and "Other setXXX Methods" on page 13-13

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

```
// Prepare to insert new names in the EMP table
PreparedStatement pstmt =
  conn.prepareStatement ("insert into EMP (EMPNO, ENAME) values (?, ?)");

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

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

// 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. 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 B-1.

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.

Closing the Connection

You must close the connection to the database after you have performed all the required operation 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.*;
import oracle.jdbc.pool.OracleDataSource;

class JdbcTest {
   public static void main (String args []) throws SQLException {
      // Create DataSource and connect to the local database
      OracleDataSource ods = new OracleDataSource();
      ods.setURL("jdbc:oracle:thin:@//myhost:1521/orcl");
      ods.setUser("scott");
      ods.setPassword("tiger");
      Connection conn = ods.getConnection();

      // Query the employee names
      Statement stmt = conn.createStatement ();
      ResultSet rset = stmt.executeQuery ("SELECT ename FROM emp");
      // Print the name out
      while (rset.next ())
         System.out.println (rset.getString (1));

      //close the result set, statement, and the connection
      rset.close();
      stmt.close();
      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 the Oracle JDBC drivers support the following kinds of stored procedures:

- PL/SQL Stored Procedures
- Java Stored Procedures

PL/SQL Stored Procedures

Oracle JDBC drivers support the processing of PL/SQL stored procedures and anonymous blocks. They support Oracle PL/SQL block syntax and most of SQL92 escape syntax. The following PL/SQL calls would work with any Oracle JDBC driver:

```
// SQL92 syntax
CallableStatement cs1 = conn.prepareCall
                      ( "{call proc (?,?)}" ) ; // stored proc
```

```
CallableStatement cs2 = conn.prepareCall
                        ( "{? = call func (?,?)}" ) ; // stored func
// Oracle 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:

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

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

Java Stored Procedures

You can use JDBC to call Java stored procedures through the SQL and PL/SQL engines. 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.

Processing SQL Exceptions

To handle error conditions, the Oracle JDBC drivers throws 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.

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, where available.

See Also:

- Appendix C, "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()`

  For errors originating in the JDBC driver, this method returns the error message with no prefix. For errors originating in the database, the method returns the error message prefixed with the corresponding `ORA` number.

- `getErrorCode()`

  For errors originating in either the JDBC driver or the database, this method returns the five-digit `ORA` number.

- `getSQLState()`

  For errors originating in the JDBC driver, this method returns no useful information. For errors originating in the database, this method returns a five-digit code indicating the SQL state. Your code should be prepared to handle `null`.

The following example prints output from a `getMessage()` 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.driver.DatabaseError.throwSqlException(DatabaseError.java:112)
at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java:146)
at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java:208)
at oracle.jdbc.driver.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, which Oracle Database 10*g* supports. It also includes chapters that cover features specific to JDBC Thin driver, JDBC Oracle Call Interface (OCI) driver, and server-side internal driver.

Part II contains the following chapters:

- Chapter 4, "JDBC Standards Support"
- Chapter 5, "Oracle Extensions"
- Chapter 6, "Features Specific to JDBC Thin"
- Chapter 7, "Features Specific to JDBC OCI"
- Chapter 8, "Server-Side Internal Driver"

4

JDBC Standards Support

Oracle Java Database Connectivity (JDBC) supports several different versions of JDBC, including JDBC 2.0 and 3.0. This chapter provides an overview of JDBC 2.0 and JDBC 3.0 support in the Oracle JDBC drivers. The chapter contains the following sections:

- Introduction
- JDBC 2.0 Support: JDK 1.2.x and Later Versions
- JDBC 3.0 Support: JDK 1.4 and Previous Releases
- Overview of Supported JDBC 3.0 Features
- Transaction Savepoints
- Retrieval of Auto-Generated Keys
- JDBC 3.0 LOB Interface Methods
- Result Set Holdability

Introduction

The Oracle JDBC drivers support all JDBC 3.0 features. These features are provided through the `oracle.jdbc` and `oracle.sql` packages. These packages support all Java Development Kit (JDK) releases from 1.2 through 1.4. JDBC 3.0 features that depend on JDK1.4 are made available to earlier JDK versions through Oracle extensions.

JDBC 2.0 Support: JDK 1.2.x and Later Versions

Standard JDBC 2.0 features are supported by JDK 1.2 and later versions. There are three areas to consider:

- Support for data types, such as objects, arrays, and large objects (LOBs). This is handled through the standard `java.sql` package.
- Support for standard features, such as result set enhancements and update batching. This 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:

- Data Type Support

- Standard Feature Support
- Extended Feature Support
- Standard versus Oracle Performance Enhancement APIs

Note: JDK1.1.x is no longer supported. The package `oracle.jdbc2` has been removed.

Data Type Support

Oracle JDBC fully supports JDK 1.2.x, 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 1.2.x environment, using the JDBC classes in `classes12.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 the Oracle Database.

Standard versus Oracle Performance Enhancement APIs

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

- Update batching
- Fetch size or row prefetching

In each case, you have the option of using the standard model or the Oracle model. 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 25-1
- "Fetch Size" on page 19-15
- "Oracle Row Prefetching" on page 25-15

JDBC 3.0 Support: JDK 1.4 and Previous Releases

Table 4–1 lists the interfaces and classes added or extended to support specific JDBC features.

Table 4–1 JDBC 3.0 Feature Support

New feature	JDK1.4 implementation	Pre-JDK1.4 implementation
Savepoints (new class)	`java.sql.Savepoint`	`oracle.jdbc.OracleSavepoint`
Savepoints (connection extensions)	`java.sql.connection`	`oracle.jdbc.OracleConnection`
Querying parameter capacities (new class)	`java.sql.ParameterMetaData`	`oracle.jdbc.OracleParameterMetaData`
Querying parameter capacities (interface change)	Not applicable	`oracle.jdbc.OraclePreparedStatement`
Resource adapters	`oracle.jdbc.connector`	`oracle.jdbc.connector`
RowSets	`oracle.jdbc.rowset`	`oracle.jdbc.rowset`
LOB modification	Not applicable	`oracle.sql.BLOB` `oracle.sql.CLOB`
Retrieving auto-generated keys	`oracle.sql.Statement`	`oracle.jdbc.OracleStatement` `oracle.jdbc.OracleConnection`
Result set holdability	`java.sql.Connection`	`oracle.jdbc.OracleConnection`
	`java.sql.DatabaseMetaData`	`oracle.jdbc.OracleDatabaseMetadata`
	`java.sql.Statement`	`oracle.jdbc.OracleStatement`

Overview of Supported JDBC 3.0 Features

Table 4–2 lists the JDBC 3.0 features supported at this release and gives references to a detailed discussion of each feature.

Table 4–2 Key Areas of JDBC 3.0 Functionality

Feature	Comments and References
Transaction savepoints	See "Transaction Savepoints" on page 4-3 for information.
Statement caching	Reuse of prepared statements by connection pools. See Chapter 22, "Statement Caching".
Switching between local and global transactions	See "Switching Between Global and Local Transactions" on page 29-4.
LOB modification	See "JDBC 3.0 LOB Interface Methods" on page 4-8.
Named SQL parameters	See "Interface oracle.jdbc.OracleCallableStatement" on page 5-21 and "Interface oracle.jdbc.OraclePreparedStatement" on page 5-20.
RowSets	See Chapter 20, "JDBC RowSets"
Retrieving auto-generated keys	See "Retrieval of Auto-Generated Keys" on page 4-6
Result set holdability	See "Result Set Holdability" on page 4-8

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 `SQLException` to be thrown.

JDK1.4 specifies a standard savepoint API. Oracle JDBC provides the following different savepoint interfaces:

- `java.sql.Savepoint`

  Works with JDK1.4

- `oracle.jdbc.OracleSavepoint`

  Works across all supported JDK versions.

JDK1.4 adds savepoint-related APIs to `java.sql.Connection`. The Oracle JDK version-independent interface, `oracle.jdbc.OracleConnection`, provides equivalent functionality.

Creating a Savepoint

You create a savepoint using either `Connection.setSavepoint`, which returns a `java.sql.Savepoint` instance, or `OracleConnection.oracleSetSavepoint`, which returns an `oracle.jdbc.OracleSavepoint` 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 `getSavepointName()`. You retrieve an ID using `getSavepointId()`.

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

Rolling back to a Savepoint

You roll back to a savepoint using `Connection.rollback(Savepoint svpt)` or `OracleConnection.oracleRollback(OracleSavepoint svpt)`. If you try to roll back to a savepoint that has been released, then `SQLException` is thrown.

Releasing a Savepoint

You remove a savepoint using one of the following methods:

- `Connection.releaseSavepoint(Savepoint svpt)`
- `OracleConnection.oracleReleaseSavepoint(OracleSavepoint svpt)`

> **Note:** As of Oracle Database 10*g*, `releaseSavepoint` and `oracleReleaseSavepoint` are not supported. If you call either of the methods, then `SQLException` is thrown with the message "Unsupported feature".

Checking Savepoint Support

You query whether savepoints are supported by your database by calling `oracle.jdbc.OracleDatabaseMetaData.supportsSavepoints()`, 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 an `SQLException` 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.

Savepoint Interfaces

The following methods are used to get information from savepoints. These methods are defined within both the `java.sql.Connection` and `oracle.jdbc.OracleSavepoint` interfaces:

- `public int getSavepointId() throws SQLException;`

  Returns the savepoint ID for an unnamed savepoint. Throws `SQLException` if `self` is a named savepoint.

- `public String getSavepointName() throws SQLException;`

  Returns the name of a named savepoint. Throws `SQLException` if `self` is an unnamed savepoint.

The following methods are defined within the `java.sql.Connection` interface:

- `public Savepoint setSavepoint() throws SQLException;`

  Creates an unnamed savepoint. Throws `SQLException` on database error or if connection is in the auto-commit mode or participating in a global transaction.

- `public Savepoint setSavepoint(String name) throws SQLException;`

  Creates a named savepoint. If a savepoint by this name already exists, then this instance replaces it. Throws `SQLException` on database error or if connection is in the auto-commit mode or participating in a global transaction.

- `public void rollback(Savepoint savepoint) throws SQLException;`

  Removes specified savepoint from current transaction. Any references to the savepoint after it is removed cause an `SQLException` to be thrown. Throws `SQLException` on database error or if connection is in the auto-commit mode or participating in a global transaction.

- `public void releaseSavepoint(Savepoint savepoint) throws SQLException;`

  Not supported at this release. Always throws `SQLException`.

Pre-JDK1.4 Savepoint Support

The following methods are defined within the `oracle.jdbc.OracleConnection` interface. Except for using `OracleSavepoint` in the signatures, they are identical to the methods declared in the preceding section.

```
public OracleSavepoint oracleSetSavepoint() throws SQLException;
public OracleSavepoint oracleSetSavepoint(String name) throws SQLException;
public void oracleRollback(OracleSavepoint savepoint)  throws SQLException;
public void oracleReleaseSavepoint(OracleSavepoint savepoint) throws SQLException;
```

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.Statement
- java.sql.Connection

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.DatabaseMetaData

In JDBC 3.0, the `java.sql.DatabaseMetaData` interface provides the following method:

```
public boolean supportsGetGeneratedKeys();
```

The method indicates whether retrieval of auto-generated keys is supported or not by the JDBC driver and the underlying data source.

java.sql.Statement

The `java.sql.Statement` interface is enhanced with the following methods:

```
public boolean execute(String sql, int autoGeneratedKeys) throws SQLException;
public boolean execute(String sql, int[] columnIndexes) throws SQLException;
public boolean execute(String sql, String[] columnNames) throws SQLException;
public boolean executeUpdate(String sql, int autoGeneratedKeys) throws
SQLException;
public boolean executeUpdate(String sql, int[] columnIndexes) throws SQLException;
public boolean executeUpdate(String sql, String[] columnNames) throws
SQLException;
public ResultSet getGeneratedKeys() throws SQLException;
```

This interface provides new methods for processing SQL statements and retrieving auto-generated keys. These methods take a String object that contains a SQL statement. They also take either the flag, `Statement.RETURN_GENERATED_KEYS`, indicating whether any generated columns are to be returned, or an array of column names or indexes specifying the columns that should be returned. The flag and the array values are considered only when an `INSERT` statement is processed. If an `UPDATE` or `DELETE` statement is processed, these values are ignored.

The `getGeneratedKeys()` method enables you to retrieve the auto-generated key fields. The auto-generated keys are returned as a ResultSet object.

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 retrieved other columns.

java.sql.Connection

The `java.sql.Connection` interface is enhanced with the following methods:

```
public PreparedStatement prepareStatement(String sql, int autoGeneratedKeys)
throws SQLException;
public PreparedStatement prepareStatement(String sql, int[] columnIndexes) throws
SQLException;
public PreparedStatement prepareStatement(String sql, String[] columnNames) throws
SQLException;
```

These methods enable you to create a `PreparedStatement` object that is capable of returning auto-generated keys.

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 in to 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.setFormOfUse(1, OraclePreparedStatement.FORM_NCHAR);
pstmt.setString(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 generated the next value. An `OraclePreparedStatement` object is created to insert a row in to the `ORDERS` table.

JDBC 3.0 LOB Interface Methods

Before Oracle Database 10*g* release 2 (10.2), Oracle provided proprietary interfaces for modification of LOB data. JDBC 3.0 adds methods for these operations. In Oracle9*i* Database release 2, the JDBC 3.0 methods were present in `ojdbc14.jar`, but were not functional. The JDBC 3.0 standard LOB methods differ slightly in name and function from the Oracle proprietary ones. In Oracle Database 10*g* release 2 (10.2), the JDBC 3.0 standard methods are implemented in both `ojdbc14.jar` and `classes12.jar`. In order to use these methods with JDK1.2 or 1.3, LOB variables must be typed as, or cast to, `oracle.sql.BLOB` or `oracle.sql.CLOB` as appropriate. With JDK1.4, LOB variables may be typed as `java.sql.Blob` or `java.sql.Clob`.

Table 4–3 and Table 4–4 show the conversions between Oracle proprietary methods and JDBC 3.0 standard methods.

Table 4–3 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)`
`trim (long len)`	`truncate(long len)`

Table 4–4 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 is a new feature introduced in 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.

The default holdability property of a `ResultSet` object is implementation defined. The default holdability of `ResultSet` objects returned by the underlying data source can be determined using the APIs provided by JDBC 3.0.

In JDBC 3.0, the following APIs have been modified to support this feature:

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

JDBC 3.0 also provides two new ResultSet constants:

- `ResultSet.HOLD_CURSORS_OVER_COMMIT`
- `ResultSet.CLOSE_CURSORS_AT_COMMIT`

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

5

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. Compared to standard JDBC, the Oracle extensions offer greater flexibility in manipulating the data. 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:

- Introduction to Oracle Extensions
- Support Features of the Oracle Extensions
- Oracle JDBC Packages
- Oracle Character Data Types Support
- Additional Oracle Type Extensions

> **Note:** This chapter focuses on type extensions, as opposed to performance extensions, which are discussed in detail in Chapter 25, "Performance Extensions".

Introduction to 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 5-4

Support Features of the Oracle Extensions

The Oracle extensions to JDBC include a number of features that enhance your ability to work with Oracle databases. Among these are support for:

- Oracle data types

- Oracle objects
- Specific schema naming
- DML returning
- Accessing PL/SQL index-by tables.

This section covers the following topics:

- Support for Oracle Data Types
- Support for Oracle Objects
- Support for Schema Naming
- DML Returning
- Accessing PL/SQL Index-by Tables

Support for Oracle Data Types

A key feature of the Oracle JDBC extensions is the type support in the `oracle.sql` package. This package includes classes that map to all the Oracle SQL data types, acting as wrappers for raw SQL data. This functionality provides two significant advantages in manipulating SQL data:

- Accessing data directly in SQL format is sometimes more efficient than first converting it to Java format.
- Performing mathematical manipulations of the data directly in SQL format avoids the loss of precision that can occur in converting between SQL and Java formats.

Once manipulations are complete and it is time to output the information, each of the `oracle.sql.*` date type support classes has all the necessary methods to convert data to appropriate Java formats.

See Also:

- Package oracle.sql on page 5-4
- "Oracle Character Data Types Support" on page 5-10
- "Additional Oracle Type Extensions" on page 5-13

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).

JDBC implementation of Oracle 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` or `oracle.sql.STRUCT` types 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.sql.ORAData` interface. Each interface specifies methods to convert data between SQL and Java.

> **Note:** The `ORAData` interface has replaced the `CustomDatum` interface. While the latter interface is deprecated, it is still supported for backward compatibility.

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 `ORAData` implementations.

For `SQLData` 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 `ORAData` interface, instead of the `SQLData` interface, in situations where portability is not a concern. `ORAData` works more easily and flexibly in conjunction with other features of the Oracle Java platform offerings.

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

Tip:

- Chapter 15, "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 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 quoted. 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 unquoted period in the object name 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 quote the type name if the type name has a dot in it.

For example, assume that user Scott creates a type called `person.address` and then wants to use it in his session. Scott may want to skip the schema name and pass in `person.address` to the JDBC driver. In this case, if `person.address` is not quoted, then the period will be detected and the JDBC driver will mistakenly interpret `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 will not change the character case even if it is quoted.

  For example, if `Scott.PersonType` is passed to the JDBC driver as an object type name, then the JDBC driver will pass 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. DML returning provides richer 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.

See Also: "DML Returning" on page 5-26

Accessing PL/SQL Index-by Tables

The Oracle JDBC drivers enable JDBC applications to make PL/SQL calls with index-by table parameters. The Oracle JDBC drivers support PL/SQL index-by tables of scalar data types

---

Note: Index-by tables of PL/SQL records are not supported.

See Also: "Accessing PL/SQL Index-by Tables" on page 5-29

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 wrappers for SQL data. The characters are converted to Java chars and, then, to bytes in the UCS2 character set.

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, as Table 5–1 indicates, implement standard JDBC 2.0 interfaces in the `java.sql` package, as well as extending the `oracle.sql.Datum` class.

Classes of the oracle.sql Package

Table 5–1 lists the `oracle.sql` data type classes and their corresponding Oracle SQL types.

Table 5–1 Oracle Data Type Classes

Java Class	Oracle SQL Types and Interfaces Implemented
`oracle.sql.STRUCT`	`STRUCT` (objects) implements `java.sql.Struct`
`oracle.sql.REF`	`REF` (object references) implements `java.sql.Ref`
`oracle.sql.ARRAY`	`VARRAY` or nested table (collections) implements `java.sql.Array`
`oracle.sql.BLOB`	`BLOB` (binary large objects) implements java.sql.Blob
`oracle.sql.CLOB`	SQL `CLOB` (character large objects) and globalization support `NCLOB` data types both implement `java.sql.Clob`
`oracle.sql.BFILE`	`BFILE` (external files)
`oracle.sql.CHAR`	`CHAR, NCHAR, VARCHAR2, NVARCHAR2`
`oracle.sql.DATE`	`DATE`
`oracle.sql.TIMESTAMP`	`TIMESTAMP`
`oracle.sql.TIMESTAMPTZ`	`TIMESTAMP WITH TIME ZONE`
`oracle.sql.TIMESTAMPLTZ`	`TIMESTAMP WITH LOCAL TIME ZONE`
`oracle.sql.NUMBER`	`NUMBER`
`oracle.sql.RAW`	`RAW`
`oracle.sql.ROWID`	`ROWID` (row identifiers)
`oracle.sql.OPAQUE`	`OPAQUE`

> **Note:** 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.

In addition to the data type classes, the `oracle.sql` package includes the following support classes and interfaces, primarily for use with objects and collections:

- `oracle.sql.ArrayDescriptor`

  This class is used in constructing `oracle.sql.ARRAY` objects. It describes the SQL type of the array.

- `oracle.sql.StructDescriptor`

  This class is used in constructing `oracle.sql.STRUCT` objects, which you can use as a default mapping to Oracle objects in the database.

- `oracle.sql.ORAData` and `oracle.sql.ORADataFactory`

  These interfaces are used in Java classes implementing the Oracle `ORAData` scenario of Oracle object support.

- `oracle.sql.OpaqueDescriptor`

  This class is used to obtain the meta data for an instance of the `oracle.sql.OPAQUE` class.

General oracle.sql.* Data Type Support

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

- One or more constructors, typically with a constructor that uses raw bytes as input and a constructor that takes a Java type as input
- 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 `ROWID`. The driver returns the object in the corresponding `oracle.sql.*` format. For example, it returns an Oracle ROWID as an `oracle.sql.ROWID`.

- 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

For any given Oracle object type, it is usually desirable to define a custom mapping between SQL and Java. For example, if you use a `SQLData` custom Java class, then the mapping must be defined in a type map.

If you choose not to define a mapping, however, then data from the object type will be materialized in Java in an instance of the `oracle.sql.STRUCT` class.

The `STRUCT` class implements the standard JDBC 2.0 `java.sql.Struct` interface and extends the `oracle.sql.Datum` class.

A `STRUCT` object is a Java representation of the raw bytes of an Oracle object. It contains the SQL type name of the Oracle object and an array of `oracle.sql.Datum` objects that hold the attribute values in SQL format.

The signature of the constructors for `STRUCT` are as follows:

```
STRUCT(Connection connection, java.sql.StructDescriptor structDescriptor, Object[]
attributes)

STRUCT(Connection connection, java.sql.StructDescriptor structDescriptor,
```

```
java.util.Map map)
```

You can materialize attributes of a `STRUCT` object as `oracle.sql.Datum[]` objects, if you use the `getOracleAttributes` method, or as `java.lang.Object[]` objects, if you use the `getAttributes` method. Materializing the attributes as `oracle.sql.*` objects gives you the following advantages of the `oracle.sql.*` format:

- Materializing `oracle.sql.STRUCT` data in `oracle.sql.*` format completely preserves data by maintaining it in SQL format. No translation is performed. This is useful if you want to access data but not necessarily display it.
- It allows complete flexibility in how your Java application unpacks data.

Notes:

- Elements of the array, although of the generic `Datum` type, actually contain data associated with the relevant `oracle.sql.*` type appropriate for the given attribute. You can cast the element to the appropriate `oracle.sql.*` type as desired. For example, a `CHAR` data attribute within the `STRUCT` is materialized as `oracle.sql.Datum`. To use it as `CHAR` data, you must cast it to `oracle.sql.CHAR`.
- Nested objects in the values array of a `STRUCT` object are materialized by the JDBC driver as instances of `STRUCT`.

In some cases, you may want to manually create a `STRUCT` object and pass it to a prepared statement or callable statement. To do this, you must also create a `StructDescriptor` object.

> **See Also:** "Using the Default STRUCT Class for Oracle Objects" on page 15-2

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. It implements the standard JDBC 2.0 `java.sql.Ref` 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.

> **See Also:** Chapter 17, "Using Oracle Object References".

Overview of Class oracle.sql.ARRAY

The `oracle.sql.ARRAY` class supports Oracle collections, either VARRAYs or nested tables. If you select either a VARRAY or a nested table from the database, then the JDBC driver materializes it as an object of the `ARRAY` class. The structure of the data is equivalent in either case. The `oracle.sql.ARRAY` class extends `oracle.sql.Datum` and implements the standard JDBC 2.0 `java.sql.Array` interface.

You can use the `setARRAY` method of the `OraclePreparedStatement` or `OracleCallableStatement` class to pass an `ARRAY` as an input parameter to a prepared statement. Similarly, you might want to manually create an `ARRAY` object to pass it to a prepared statement or callable statement, perhaps to insert into the database. This involves the use of `ArrayDescriptor` objects.

See Also: "Overview of Collection Functionality" on page 18-3

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.

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.

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 16, "Working with LOBs and BFILEs".

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

These classes map to primitive SQL data types, which are a part of standard JDBC, and supply conversions to and from the corresponding JDBC Java types.

Because Java `Double` and `Float` `NaN` values do not have an equivalent Oracle `NUMBER` representation, a `NullPointerException` is thrown whenever a `Double.NaN` value or a `Float.NaN` value is converted into an Oracle `NUMBER` using `oracle.sql.NUMBER`, 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
```

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 from Sun Microsystems. Time zones are specified by using the `java.util.Calendar` class.

Note: Do not use `TimeZone.getTimeZone` to create time zone objects. The Oracle time zone data types support more time zone names than does the JDK.

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

Overview of Class oracle.sql.ROWID

This class supports Oracle ROWIDs, which are unique identifiers for rows in database tables. You can select a ROWID as you would select any column of data from the table. Note, however, that you cannot manually update ROWIDs. The Oracle Database updates them automatically as appropriate.

The `oracle.sql.ROWID` class does not implement any noteworthy functionality beyond what is in the `oracle.sql.Datum` superclass. However, `ROWID` does provide a `stringValue` method that overrides the `stringValue` method in the `oracle.sql.Datum` class and returns the hexadecimal representation of the `ROWID` bytes.

See Also: "Oracle ROWID Type" on page 5-14

Class oracle.sql.OPAQUE

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

Note: There is minimal support for OPAQUE type.

The following are the methods of the `oracle.sql.OPAQUE` class:

- `getBytesValue`

  Returns a `byte` array that represents the value of the `OPAQUE` object, in the format used in the database.

- `isConvertibleTo(Class jClass)`

Determines if a `Datum` object can be converted to a particular class, where `Class` is any class and `jClass` is the class to convert. `true` is returned if conversion to `jClass` is permitted and, `false` is returned if conversion to `jClass` is not permitted.

- `getDescriptor`

  Returns the `OpaqueDescriptor` object that contains the type information.

- `getJavaSqlConnection`

  Returns the connection associated with the receiver. Because methods that use the `oracle.jdbc.driver` package are deprecated, the `getConnection` method has been deprecated in favor of the `getJavaSqlConnection` method.

- `getSQLTypeName`

  Implements the `java.sql.Struct` interface function and retrieves the SQL type name of the SQL structured type that this `Struct` object represents. This method returns the fully-qualified type name of the SQL structured type which this `STRUCT` object represents.

- `getValue`

  Returns a Java object that represents the value.

- `toJdbc`

  Returns the JDBC representation of the `Datum` object.

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 5-17

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
- SQL NCHAR Data Types
- Class oracle.sql.CHAR

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 allow 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 only difference in usage between the two data types occur in a data bind situation: a JDBC program must call the `setFormOfUse` method to specify if the data is bound for a SQL NCHAR data type.

> **Note:** The `setFormOfUse` method must be called before the `registerOutParameter` method is called in order to avoid unpredictable results.

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.setFormOfUse(2, OraclePreparedStatement.FORM_NCHAR);
pstmt.setFormOfUse(3, OraclePreparedStatement.FORM_NCHAR);

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

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 the Oracle JDBC drivers prior to 10*g* release 1 (10.1), there were performance advantages to using oracle.SQL.CHAR. In Oracle Database 10*g*, there are no longer any such advantages. In fact, optimum performance is achieved using java.lang.String. All Oracle JDBC drivers handle all character data in the Java UCS2 character set. Using oracle.sql.CHAR does not prevent conversions between the database character set and UCS2.

The only remaining use of oracle.sql.CHAR 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 java.lang.String. 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 as needed.

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);
```

The `CHAR` class has multiple constructors, 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.

Notes:

- The `CharacterSet` object cannot be null.
- 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`

  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 throws a `SQLException`.

- `toString`

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

- `getStringWithReplacement`

  Identical to `getString`, 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 21, "Globalization Support"

Additional Oracle Type Extensions

This section covers 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. 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
- Oracle REF CURSOR Type Category
- Oracle BINARY_FLOAT and BINARY_DOUBLE Types
- The oracle.jdbc Package

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 wrapper 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 updation, as in the example that follows.

> **Note:** The `oracle.sql.ROWID` class replaces `oracle.jdbc.driver.ROWID`, which was used in previous releases of Oracle JDBC.

Example

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

```
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 ename, rowid FROM emp FOR UPDATE");

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

PreparedStatement pstmt =
   conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");

// Loop through the results of the query
while (rset.next ())
{
    String ename = rset.getString (1);
    oracle.sql.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 ename from emp; end;");

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

// Use the cursor like a normal 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 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` type column, the number bits are returned and converted to the Java `double` type.

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

A call to the JDBC standard `setDouble(int, double)` method of `PreparedStatement` converts the Java `double` argument to Oracle `NUMBER` style bits and send them to the database. In contrast, the `setBinaryDouble(int, double)` method of `oracle.jdbc.OraclePreparedStatement` 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 `PreparedStatement`. 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 use excess CPU 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 about BINARY_DOUBLE, the same is true for BINARY_FLOAT.

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 Oracle 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 5–2 lists key interfaces and classes used for connections, statements, and result sets.

Table 5–2 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 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	Provide set*XXX* methods to bind oracle.sql.* types into a prepared statement
		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

Table 5–2 (Cont.) Key Interfaces and Classes of the oracle.jdbc Package

Name	Interface or Class	Key Functionality
`OracleResultSet`	Interface	Provides `get`*`XXX`* methods to retrieve data in `oracle.sql` format Implements `java.sql.ResultSet`
`OracleResultSetMetaData`	Interface	Provides methods to get meta information about Oracle result sets, such as column names and data types Implements `java.sql.ResultSetMetaData`
`OracleDatabaseMetaData`	Class	Provides methods to get meta information about the database, such as database product name/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.

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
- Method getJavaSqlConnection

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.

Client Identifiers

In a connection pooling environment, the client identifier can be used to identify which light-weight user is currently using the database session. 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 Application Developer's Guide - Fundamentals*

Key methods include:

- `createStatement`

  Allocates a new `OracleStatement` object

- `prepareStatement`

  Allocates a new `OraclePreparedStatement` object

- `prepareCall`

  Allocates a new `OracleCallableStatement` object

- `getTypeMap`

  Retrieves the type map for this connection, for use in mapping Oracle object types to Java classes

- `setTypeMap`

  Initializes or updates the type map for this connection, for use in mapping Oracle object types to Java classes

- `getTransactionIsolation`

  Gets this connection's current isolation mode

- `setTransactionIsolation`

  Changes the transaction isolation level using one of the `TRANSACTION_*` values

The following `oracle.jdbc.OracleConnection` methods are Oracle-defined extensions:

- `setClientIdentifier`

  Sets the client identifier for this connection

- `clearClientIdentifier`

  Clears the client identifier for this connection

- `getDefaultExecuteBatch`

  Retrieves the default update-batching value for this connection

- `setDefaultExecuteBatch`

  Sets the default update-batching value for this connection

- `getDefaultRowPrefetch`

  Retrieves the default row-prefetch value for this connection

- `setDefaultRowPrefetch`

  Sets the default row-prefetch value for this connection

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 connection-wide basis.

Key methods include:

- `executeQuery`

Runs a database query and returns an `OracleResultSet` object

- `getResultSet`

  Retrieves an `OracleResultSet` object

- `close`

  Closes the current statement

The following `oracle.jdbc.OracleStatement` methods are Oracle-defined extensions:

- `defineColumnType`

  Defines the type you will use to retrieve data from a particular database table column

  > **Note:** This method is no longer needed or recommended for use with the JDBC Thin driver.

- `getRowPrefetch`

  Retrieves the row-prefetch value for this statement

- `setRowPrefetch`

  Sets the row-prefetch value for this statement

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 `set`*`XXX`* methods for binding `oracle.sql.*` types and objects to prepared statements, and methods to support Oracle performance extensions on a statement-by-statement basis.

> **Note:** Do not use `PreparedStatement` 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`

Key methods include:

- `getExecuteBatch`

  This method retrieves the update-batching value for this statement.

- `setExecuteBatch`

  This method sets the update-batching value for this statement.

- `setOracleObject`

  This is a generic `set`*`XXX`* method for binding `oracle.sql.*` data to a prepared statement as an `oracle.sql.Datum` object.

- `set`*`XXX`*

These methods, such as `setBLOB`, are for binding specific `oracle.sql.*` types to prepared statements.

- `set`*`XXX`*`AtName`

  Unlike the JDBC standard method `set`*`XXX`*`(int, `*`XXX`*`)`, which sets the value of the *n*th SQL parameter specified by the integer argument, `set`*`XXX`*`AtName(String, `*`XXX`*`)` sets the SQL parameter with the specified character name in the SQL string. The SQL parameter is a SQL identifier preceded by a colon (`:`). For example, `:id` in:

  ```
  ps = conn.prepareStatement("select * from tab where id = :id");
    ((OraclePreparedStatement)ps).setIntAtName("id", 42);
  ```

- `setORAData`

  This method binds an `ORAData` object to a prepared statement.

- `setNull`

  This method sets the value of the object specified by its SQL type name to `NULL`. For `setNull(`*`param_index`*`, `*`type_code`*`, `*`sql_type_name`*`)`, if *`type_code`* is `REF`, `ARRAY`, or `STRUCT`, then *`sql_type_name`* is the fully qualified name of the SQL type.

- `setFormOfUse`

  This method sets which form of use this method is going to use. There are two constants that specify the form of use: `FORM_CHAR` and `FORM_NCHAR`, where `FORM_CHAR` is the default, meaning that the regular database character set is used. If the form of use is set to `FORM_NCHAR`, the JDBC driver will represent the provided data in the national character set of the server. The following code shows how the `FORM_NCHAR` is used:

  ```
  pstmt.setFormOfUse (parameter index,
  oracle.jdbc.OraclePreparedStatement.FORM_NCHAR)
  ```

- `close`

  This method closes the current statement.

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 `CallableStatement` 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`

Key methods include:

- `getOracleObject`

  This is a generic `get`*`XXX`* method for retrieving data into an `oracle.sql.Datum` object, which can be cast to the specific `oracle.sql.*` type as necessary.

- `get`*`XXX`*

These methods, such as `getCLOB`, are for retrieving data into specific `oracle.sql.*` objects.

- `setOracleObject`

  This is a generic `set`*`XXX`* method for binding `oracle.sql.*` data into a callable statement as an `oracle.sql.Datum` object.

- `set`*`XXX`*

  These methods, such as `setBLOB`, are inherited from `OraclePreparedStatement` for binding specific `oracle.sql.*` objects into callable statements.

- `set`*`XXX`*`(String, `*`XXX`*`)`

  The definition of a PL/SQL stored procedure may include one or more named parameters. When you create a `CallableStatement` to call this stored procedure, you must supply values for all IN parameters. You can either do this with the JDBC standard `set`*`XXX`*`(int, `*`XXX`*`)` methods, or using the Oracle extension `setXXX(String, `*`XXX`*`)`. The first argument to this method specifies the name of the PL/SQL formal parameter and the second argument specifies the value. For example, consider a stored procedure `foo`, defined as follows:

```
CREATE OR REPLACE PROCEDURE foo (myparameter VARCHAR2) IS
  BEGIN
  ...
  END;
```

  Create an `OracleCallableStatement` to call `foo`, as follows:

```
OracleCallableStatement cs = (OracleCallableStatement)
   conn.prepareCall("{call foo(?)}");
```

  You can pass the string "bar" to this procedure in one of the following two ways:

```
cs.setString(1,"bar"); // JDBC standard
// or...
cs.setString("myparameter","bar"); // Oracle extension
```

  Note: The argument is the name of the formal parameter declared in the PL/SQL stored procedure. This name does not necessarily appear anywhere in the SQL string. This differs from the `set`*`XXX`*`AtName` method, whose first argument is a substring of the SQL string.

- `set`*`XXX`*`AtName`

  Unlike the JDBC standard method `set`*`XXX`*`(int, `*`XXX`*`)`, which sets the value of the *n*th SQL parameter specified by the integer argument, `set`*`XXX`*`AtName(String, `*`XXX`*`)` sets the SQL parameter with the specified character name in the SQL string. The SQL parameter is a SQL identifier preceded by a colon (`:`). For example, `:id` in:

```
OracleCallableStatement cs = (OracleCallableStatement)
      conn.prepareCall("call foo(:id)");
cs.setIntAtName("id", 42);
```

- `setNull`

This method sets the value of the object specified by its SQL type name to NULL. For setNull(*param_index*, *type_code*, *sql_type_name*), if *type_code* is REF, ARRAY, or STRUCT, then *sql_type_name* is the fully qualified name of the SQL type.

- setFormOfUse

  This method sets which form of use this method is going to use. There are two constants that specify the form of use: FORM_CHAR and FORM_NCHAR, where FORM_CHAR is the default. If the form of use is set to FORM_NCHAR, then the JDBC driver will represent the provided data in the national character set of the server. The following code show how FORM_NCHAR is used:

  ```
  pstmt.setFormOfUse (parameter index,
  oracle.jdbc.OraclePreparedStatement.FORM_NCHAR)
  ```

- registerOutParameter

  This method registers the SQL type code of the output parameter of the statement. JDBC requires this for any callable statement with an OUT parameter. It takes an integer parameter index, the position of the output variable in the statement, relative to the other parameters, and an integer SQL type, the type constant defined in oracle.jdbc.OracleTypes.

  This is an overloaded method. One version of this method is for named types only, when the SQL type code is OracleTypes.REF, STRUCT, or ARRAY. In this case, in addition to a parameter index and SQL type, the method also takes a String SQL type name, the name of the Oracle user-defined type in the database, such as EMPLOYEE.

- close

  This method closes the current result set, if any, and the current statement.

Notes:

- The set*XXX*(String,...) and registerOutParameter(String,...) methods can only be used if all binds are procedure or function parameters only. The statement can contain no other binds and the parameter binds must be indicated with ? and not :*XX*.
- If you are using set*XXX*(int,...), set*XXX*AtName(String,...) or a combination of both, 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 get*XXX* methods for retrieving data into oracle.sql.* objects.

Key methods include:

- getOracleObject

  This is a generic get*XXX* method for retrieving data into an oracle.sql.Datum object. It can be cast to the specific oracle.sql.* type as necessary.

- `get`*`XXX`*

  These methods, such as `getCLOB`, are 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 Meta Data Extensions" on page 13-17

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)

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

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 `registerOutputParameter` can take for `CallableStatement` and `OracleCallableStatement`:

```
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` 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 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 setNull can take for 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 PreparedStatement to insert a NULL 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 employee_table VALUES (?)");

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

Method getJavaSqlConnection

The getJavaSqlConnection method of the oracle.sql.* classes returns java.sql.Connection while the getConnection method returns

`oracle.jdbc.driver.OracleConnection`. Because the methods that use the `oracle.jdbc.driver` package are deprecated, the `getConnection` method is also deprecated in favor of the `getJavaSqlConnection` method.

For the following Oracle data type classes, the `getJavaSqlConnection` method is available:

- `oracle.sql.ARRAY`
- `oracle.sql.BFILE`
- `oracle.sql.BLOB`
- `oracle.sql.CLOB`
- `oracle.sql.OPAQUE`
- `oracle.sql.REF`
- `oracle.sql.STRUCT`

The following shows the `getJavaSqlConnection` and the `getConnection` methods in the `Array` class:

```
public class ARRAY
{
  // New API
  //
  java.sql.Connection getJavaSqlConnection()
    throws SQLException;

  // Deprecated API.
  //
  oracle.jdbc.driver.OracleConnection
    getConnection() throws SQLException;

  ...
}
```

DML Returning

DML returning provides richer 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.

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 4-6

Oracle-Specific APIs

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

```
public void registerReturnParameter(int paramIndex, int externalType) throws
SQLException;
public void registerReturnParameter(int paramIndex, int externalType, int maxSize)
throws SQLException;
public void registerReturnParameter(int paramIndex, intsqlType, String typeName)
throws SQLException;
public ResultSet getReturnResultSet() throws SQLException;
```

The registerReturnParameter method is used to register the return parameter for DML returning. The method throws an SQLException instance if an error occurs. The paramIndex parameter is used to specify the index of the return parameter. Its value should be greater than zero. The externalType parameter specifies the type of the return parameter. The maxSize parameter specifies the maximum bytes or characters of the return parameter. This method can be used only with char or RAW types. The typeName parameter specifies the fully-qualified name of a SQL structured 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 if an error occurs.

> **Note:** The Oracle-specific API for DML returning are in classes12.jar for Java Development Kit (JDK) 1.2.x and JDK 1.3.x and in ojdbc14.jar for JDK 1.4.x

Running DML Returning Statements

Before running a DML returning statement, the JDBC application needs to 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, you need to 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 need to 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 Index-by Tables

The Oracle JDBC drivers enable JDBC applications to make PL/SQL calls with index-by table parameters. This section covers the following topics:

- Overview
- Binding IN Parameters
- Receiving OUT Parameters
- Type Mappings

> **Note:** Index-by tables of PL/SQL records are not supported.

Overview

The Oracle JDBC drivers support PL/SQL index-by tables of scalar data types. Table 5–3 displays the supported scalar types and the corresponding JDBC type codes.

Table 5–3 PL/SQL Types and Corresponding JDBC Types

PL/SQL Types	JDBC Types
BINARY_INTEGER	NUMERIC
NATURAL	NUMERIC

Table 5–3 (Cont.) PL/SQL Types and Corresponding JDBC Types

PL/SQL Types	JDBC Types
NATURALN	NUMERIC
PLS_INTEGER	NUMERIC
POSITIVE	NUMERIC
POSITIVEN	NUMERIC
SIGNTYPE	NUMERIC
STRING	VARCHAR

Note: Oracle JDBC does not support `RAW`, `DATE`, and PL/SQL `RECORD` as element types.

Typical Oracle JDBC input binding, output registration, and data-access methods do not support PL/SQL index-by tables. This chapter introduces 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 index-by tables as `IN`, `OUT`, or `IN OUT` parameters, including function return values.

See Also: *Oracle Database PL/SQL User's Guide and Reference*

Binding IN Parameters

To bind a PL/SQL index-by table 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 5–4 describes the arguments of the `setPlsqlIndexTable` method.

Table 5–4 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 index-by table 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[]`.

Table 5–4 (Cont.) Arguments of the setPlsqlIndexTable Method

Argument	Description
`int maxLen`	Specifies the maximum table length of the index-by table 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 index-by table 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 index-by table element type based on the values defined in the `OracleTypes` class.
`int elemMaxLen`	Specifies the index-table 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 index-by table as an `IN` parameter:

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

// index-by table bind value
int[] values = { 1, 2, 3 };

// maximum length of the index-by table 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 index-by table bind value
int currentLen = values.length;

// index-by table element type
int elemSqlType = OracleTypes.NUMBER;

// index-by table 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 index-by table 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 index-by table 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 5–5 describes the arguments of the `registerIndexTableOutParameter` method.

Table 5–5 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 index-by table bind value to be returned.
`int elemSqlType`	Specifies the index-by table element type based on the values defined in the `OracleTypes` class.
`int elemMaxLen`	Specifies the index-by table 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 index-by table as an `OUT` parameter:

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

// index-by table element type
int elemSqlType = OracleTypes.NUMBER;

// index-by table 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 index-by table values in different mapping styles. There are three mapping choices available in JDBC drivers:

Mappings	Methods to Use
JDBC default mappings	`getPlsqlIndexTable(int)`

Mappings	Methods to Use
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 with returns index-by table elements using the JDBC default mappings. The syntax for this method is:

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

Table 5–6 describes the argument of the `getPlsqlIndexTable` method.

Table 5–6 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 index-by table with elements of `NUMERIC` type code, the element values are mapped to `BigDecimal` by the 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 index-by table 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 index-by table elements using Oracle mapping.

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

Table 5–7 describes the argument of the `getOraclePlsqlIndexTable` method.

Table 5–7 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 index-by table 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 index-by table `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 index-by table elements in Java primitive types. The return value is a Java array. The syntax for this method is:

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

Table 5–8 describes the arguments of the `getPlsqlIndexTable` method.

Table 5–8 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 index-by table 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 index-by table 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 index-by table value. This
// value defines the maximum table size to be returned.
int maxLen = 10;

// index-by table element type
int elemSqlType = OracleTypes.NUMBER;

// index-by table 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]);
```

6

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 on working with Oracle JDBC applets. This following topics are covered in this chapter:

- Introduction to JDBC Thin Client
- Additional Features Supported
- JDBC in Applets

Introduction to 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 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 client side and server side. On the client side, you use this driver to access application servers, from a Java method to access data on Oracle Database, and to access applets. On the server side, this driver is used to access a remote Oracle Database instance or another session on the same database. Typically, on the server side, this driver is used to access the Oracle Database from a Java stored procedure on the 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:

- Support for Applets
- Default Support for Native XA

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 normally 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"

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 29-20

JDBC in Applets

This section describes what you must do for the applet to connect to a database, including how to use the Oracle Connection Manager or signed applets if you are connecting to a database not running on the same host as 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:sid` or in the form of a TNS keyword-value syntax.

For example, if the database to which you want to connect resides on host `prodHost`, at port `1521`, and SID `ORCL`, and you want to connect with user name `scott` with password `tiger`, then use either of the two following connect strings:

using `host:port:sid` syntax:

```
String connString="jdbc:oracle:thin:@prodHost:1521:ORCL";

OracleDataSource ods = new OracleDataSource();
ods.setURL(connString);
ods.setUser("scott");
ods.setPassword("tiger");
Connection conn = ods.getConnection();
```

using TNS keyword-value syntax:

```
String connString = "jdbc:oracle:thin:@(description=(address_list=
   (address=(protocol=tcp)(port=1521)(host=prodHost)))
OracleDataSource ods = new OracleDataSource();

ods.setURL(connString);
ods.setUser("scott");
ods.setPassword("tiger");
Connection conn = ods.getConnection();
   (connect_data=(INSTANCE_NAME=ORCL)))";
```

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. There are two possible ways to make your application work with this important security restriction:

- Connect to the database by using the Oracle Connection Manager.
- Use a signed applet to connect to the database directly.

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

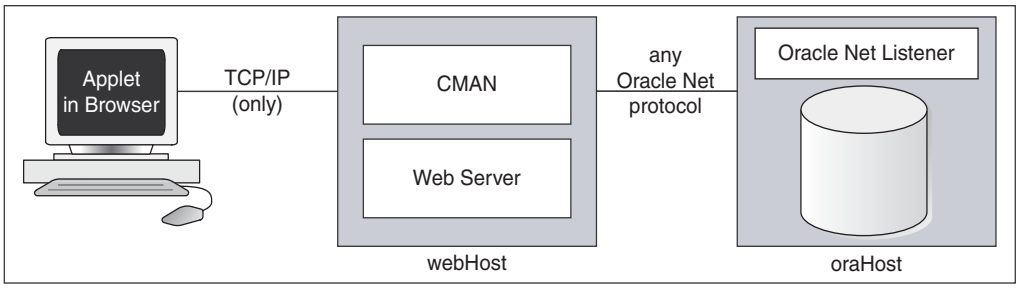
Using the Oracle Connection Manager

The Oracle Connection Manager is a lightweight, highly-scalable program that can receive Oracle Net packets and re-transmit 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 6–1 illustrates the relationship between the applet, the Oracle Connection Manager, and the database.

Figure 6–1 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 connect 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
```

To use your applet, you must now write the connect 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 6–1. 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 `1521`, and SID `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=1521)(host=oraHost)))
   (connect_data=(INSTANCE_NAME=orcl))
   (source_route=yes))";
   OracleDataSource ods = new OracleDataSource();
   ods.setURL(myURL);
   ods.setUser("scott");
   ods.setPassword("tiger");
   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. In 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 on the steps you must follow to sign an applet, refer to the Signed Applet Example at:

   http://java.sun.com/developer/technicalArticles/Security/Signed/index.html

2. Include applet code that asks for appropriate permission 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:scott/tiger@dlsun511:1721:orcl");
   Connection conn = ods.getConnection();
   ```

3. You must obtain an object-signing certificate. Refer to the Object-Signing Resources page at:

   http://developer.netscape.com/software/signedobj/index.html

This site provides information on obtaining and installing a certificate.

For more information on writing applet code that asks for permissions, refer to Introduction to Capabilities Classes at:

http://developer.netscape.com/docs/manuals/signedobj/capabilities/contents.htm

For information about the Java Security API, including signed applet examples, see the following Sun Microsystems site:

http://java.sun.com/security

Using Applets with Firewalls

Under normal 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 hostname of the client with the rules and, based on this comparison, either grant the client access or deny access. If the hostname 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 hostnames 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 hostname or environment variables on the local system. As a result, the JDBC Thin driver cannot access the hostname on which it is running. The firewall cannot be provided with the hostname. 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 hostname, of the host on which the JDBC applet is running.

- Ensure that the hostname, "__jdbc__", never appears in the firewall rules. This hostname has been hard-coded as a false hostname inside the driver to force an IP address lookup. If you do enter this hostname 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 `1521`, with SID `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=1521)))" +
      "(source_route=yes)" +
      "(connect_data=(SERVICE_NAME=orcl)))");
);
ods.setUser("scott");
ods.setPassword("tiger");
Connection conn = ods.getConnection();
```

Note: To connect through a firewall, you cannot specify the URL in `host:port:sid` syntax. For example, a URL specified as follows will *not* work:

```
String connString =
   "jdbc:oracle:thin:@example.us.oracle.com:1521:orcl";

OracleDataSource ods = new OracleDataSource();
ods.setURL(connString);
ods.setUser("scott");
ods.setPassword("tiger");
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=1521)(host=oraHost)))
      (connect_data=(INSTANCE_NAME=orcl))
      (source_route=yes))";
OracleDataSource ods = new OracleDataSource();
ods.setURL(connString);
ods.setUser("scott");
ods.setPassword("tiger");
Connection conn = ods.getConnection();
```

When your applet uses a URL similar to the preceding URL, it will behave 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 `classes12.jar` or `ojdbc14.jar`.

Follow these steps:

1. Move the JDBC driver classes file `classes12.jar` or `ojdbc14.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 `classes12.jar` or `ojdbc14.jar` and required class files from `orai18n.jar`, 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 dialogs 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 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.

7

Features Specific to JDBC OCI

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
- 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 26, "OCI Connection Pooling"

Transparent Application Failover

The Transparent Application Failover feature of the 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.

See Also: "OCI Native XA" on page 29-20

OCI Instant Client

This section covers the following topics:

- Overview of Instant Client

- Benefits of Instant Client
- JDBC OCI Instant Client Installation Process
- Usage of Instant Client
- Patching Instant Client Shared Libraries
- Regeneration of Data Shared Library and ZIP files
- Database Connection Names for OCI Instant Client
- Environment Variables for OCI Instant Client

Overview of Instant Client

The Instant Client feature 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 running in the Instant Client mode 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 7–1 shows the Oracle client-side files required to deploy a JDBC OCI application. Library names of release 10.2 are used in the table. The number part of library names will change in future releases to agree with the release.

Table 7–1 OCI Instant Client Shared Libraries

Linux and UNIX	Description for Linux and UNIX	Windows	Description for Windows
`libclnstsh.so.10.1`	Client Code Library	`oci.dll`	Forwarding functions that applications link with
`libociei.so`	OCI Instant Client Data Shared Library	`oraociei10.dll`	Data and code
`libnnz10.so`	Security Library	`orannzsbb10.dll`	Security Library
`libocijdbc10.so`	OCI Instant Client JDBC Library	`oraocijdbc10.dll`	OCI Instant Client JDBC Library
ALL JDBC Java Archive (JAR) files	**See Also:** "Checking the Environment Variables" on page 2-3	All JDBC JAR files	**See Also:** "Checking the Environment Variables" on page 2-3

Note: To provide Native XA functionality, you must copy the JDBC XA class library. On UNIX, this library, `libheteroxa10.so`, is located in *ORACLE_HOME*`/jdbc/lib`. On Windows, this library, `heteroxa10.dll`, is located in *ORACLE_HOME*`\bin`.

Benefits of Instant Client

The benefits of Instant Client are:

- Installation involves copying a small 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 the 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, set `LD_LIBRARY_PATH` to `instantclient`. On Windows, set `PATH` to locate the `instantclient` directory.
3. Add the full pathnames of the JDBC class libraries to the `CLASSPATH` environment variable.

After completing these steps you are ready to run the JDBC OCI application.

The JDBC OCI application operates in the Instant Client mode when the OCI and JDBC shared libraries are accessible through the library path environment variable. In the Instant Client mode, there is no dependency on `ORACLE_HOME` and none of the other code and data files provided in `ORACLE_HOME` are needed by JDBC OCI, except for the `tnsnames.ora` file.

Instant Client can be also installed from the Oracle Universal Installer by selecting the Instant Client option. The installation should be done into an empty directory. As with the OTN install, you must set the `LD_LIBRARY_PATH` environment variable to the instant client directory to operate in the Instant Client mode.

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:

- `libociei.so` library is in `$ORACLE_HOME/instantclient`
- `libclnstsh.so.10.1`, `libocijdbc10.so`, and `libnnz10.so` are in `$ORACLE_HOME/lib`
- The JDBC class libraries are in `$ORACLE_HOME/jdbc/lib`

On Windows:

- `oraociei10.dll` library is in `ORACLE_HOME\instantclient`
- `oci.dll`, `oraocijdbc10.dll`, and `orannzsbb10.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 pathnames of the JDBC class libraries to `CLASSPATH`, you can enable running the JDBC OCI application in the Instant Client mode.

Notes:

- To provide Native XA functionality, you must copy the JDBC XA class library. On UNIX, this library, `libheteroxa10.so`, is located in *ORACLE_HOME*`/jdbc/lib`. On Windows, this library, `heteroxa10.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 be operated in the Instant Client mode, 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 being run in the Instant Client mode, except that the Instant Client mode is for client-side operation only. Therefore, server-side external procedures cannot operate in the Instant Client mode.

Patching Instant Client Shared Libraries

Because Instant Client is a deployment feature, the emphasis has been 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 7–1, " 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.10.1`. 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 Install 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 `libociei.so` 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 Windows platforms.

Database Connection Names for OCI Instant Client

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

- A Thin-style connect string of the form:

  host:*port*:*service_name*

  For example:

  ```
  url="jdbc:oracle:oci:@//example.com:5521:bjava21"
  ```

- A SQL Connect URL string of the form:

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

  For example:

```
url="jdbc:oracle:oci:@//example.com:5521/bjava21
```

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

```
url="jdbc:oracle:oci:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
  (HOST=dlsun242) (PORT=5521))
  (CONNECT_DATA=(SERVICE_NAME=bjava21)))"
```

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

> **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 connect string is not supported. However, an alternate way to use the empty connect string is to set the TWO_TASK environment variable on UNIX, or the LOCAL variable on Windows, to either a tnsnames.ora entry or an Oracle Net keyword-value pair. If TWO_TASK or LOCAL is set to a tnsnames.ora entry, then the tnsnames.ora file must be 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=1573)))

SID_LIST_LISTENER = (SID_LIST=
                     (SID_DESC=(SID_NAME=rdbms3)
                     (GLOBAL_DBNAME=rdbms3.server6.us.alchemy.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=1573))
                     (CONNECT_DATA=(SERVICE_NAME=rdbms3.server6.us.alchemy.com)))"
```

or:

```
url = "jdbc:oracle:oci:@//server6:1573/rdbms3.server6.us.alchemy.com"
```

Alternatively, you can set the TWO_TASK environment variable to any of the connect strings and connect to the database server without specifying the connect 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=1573))
                 (CONNECT_DATA=(SERVICE_NAME=rdbms3.server6.us.alchemy.com)))"
```

or:

```
setenv TWO_TASK //server6:1573/rdbms3.server6.us.alchemy.com
```

Now, you can connect to the database server using the following URL:

```
url = "jdbc:oracle:oci:@"
```

The connect 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=1573))
                (CONNECT_DATA=(SERVICE_NAME=rdbms3.server6.us.alchemy.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` connect string, 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 Globalization Support files 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 will always obtain its data from the Data Shared Library. If the Data Shared Library is not available, only then is the `ORACLE_HOME` environment variable 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 in the Instant Client mode.

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

On UNIX:

```
setenv ORA_TZFILE timezlrg.dat
```

On Windows:

```
set ORA_TZFILE timezlrg.dat
```

If the driver is not operating in the Instant Client mode, 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, `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 light weight 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 light weight data shared library, `libociicus.so` on UNIX platforms, which is 4 MB in size and a subset of the data shared library, `libociei.so`, which is 67 MB in size.

The light weight 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 7–2 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 7–2 Data Shared Library for Instant Client and Instant Client Light (English)

Platform	Instant Client	Instant Client Light (English)
Sun Solaris	`libociei.so` (67 MB)	`libociicus.so` (4 MB)
Linux	`libociei.so` (67 MB)	`libociicus.so` (4 MB)
Microsoft Windows	`oraociei10.dll` (85 MB)	`oraociicus10.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 operate in the Instant Client Light mode, an application needs to set the LD_LIBARARY_PATH environment variable in UNIX 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 LD_LIBRARY_PATH in UNIX or oraociei10.dll in PATH in Microsoft Windows, to determine if the application should operate in the Instant Client mode. In case this library is not found, then OCI tries to load the Instant Client Light data shared library, libociicus.so in UNIX or libociicus10.dll in

Microsoft Windows. If this library is found, then the application operates in the Instant Client Light mode. Otherwise, a non-Instant Client mode is assumed.

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 `instantclient_10_2` directory in which the light weight libraries are unzipped should be empty before the unzip.

- From Client Admin Install

  Instead of copying `libociei.so` or `oraociei10.dll` from the *ORACLE_HOME*`/instantclient directory`, copy `libociicus.so` or `oraociic10.dll` from the *ORACLE_HOME*`/instantclient/light` directory. That is, the Instant Client directory on the `LD_LIBRARY_PATH`, in UNIX, 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, `PATH` should contain `oraociicus10.dll` instead of `oraociei10.dll`.

- From Oracle Universal Installer

  If the Instant Client option is selected from the Oracle Universal Installer, then `libociei.so` (or `oraociei10.dll` on Microsoft Windows) is installed in the base directory of the installation which is going to be placed on `LD_LIBRARY_PATH`. This is so that Instant Client Light is not enabled by default. The Instant Client Light data shared library, `libociicus.so` (or `oraociicus10.dll` on Microsoft Windows), is installed in the `light` subdirectory of the base directory. Therefore, to operate in the Instant Client Light mode, the OCI data shared library, `libociei.so` (or `oraociei10.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 the Oracle Universal Installer has installed the Instant Client in `my_oraic_10_2` directory on `LD_LIBRARY_PATH`, then one would need to do the following to operate in the Instant Client Light mode.

```
cd my_oraic_10_2
rm libociei.so
mv light/libociicus.so .
```

> **Note:** All the Instant Client files should be always copied or installed in an empty directory. This is to ensure that no incompatible binaries exist in the installation.

8

Server-Side Internal Driver

This chapter covers the following topics:

- Introduction
- Connecting to the Database
- Exception-Handling Extensions
- Session and Transaction Context
- Testing JDBC on the Server
- Loading an Application into the Server
- Server-Side Character Set Conversion of oracle.sql.CHAR Data

Introduction

The server-side internal driver is intrinsically tied to the Oracle Database and to the Java virtual machine (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 JVM was started.

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 either `jdbc:oracle:kprb:` or `jdbc:default:connection:` as the URL string.

- 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 connect 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
         OracleDriver ora = new OracleDriver();
         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.

If you do call the `close` method, then be aware of the following:

- All connection instances obtained through the `defaultConnection` method, which actually reference the same database connection, will be closed and unavailable for further use, with state and resource cleanup as appropriate. Running `defaultConnection` afterward would result in a new connection object.
- Even though the connection object is closed, the implicit connection to the database will not be closed.

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();
```

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. Note that although the method is not creating a new database connection, it is returning a new object.

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.

Exception-Handling Extensions

The server-side internal driver, in addition to having standard exception-handling capabilities, such as `getMessage()`, `getErrorCode()`, and `getSQLState()`, offers extended features through the `oracle.jdbc.driver.OracleSQLException` class. This class is a subclass of the standard `java.sql.SQLException` class and is not available to the client-side JDBC drivers or the server-side Thin driver.

When an error condition occurs in the server, it often results in a series of related errors being placed in an internal error stack. The JDBC server-side internal driver retrieves errors from the stack and places them in a chain of `OracleSQLException` objects.

You can use the following methods in processing these exceptions:

- `SQLException getNextException()`

  This method returns the next exception in the chain or `null` if there are no further exceptions. You can start with the first exception you receive and work through the chain. This is a standard method.

- `int getNumParameters()` (Oracle extension)

  Errors from the server usually include parameters, or variables, that are part of the error message. These may indicate what type of error occurred, what kind of operation was being attempted, or the invalid or affected values. This method returns the number of parameters included with this error. It is an Oracle extension.

- `Object[] getParameters()` (Oracle extension)

  This method returns a Java `Object[]` array containing the parameters included with this error. It is an Oracle extension.

Example

Following is an example of server-side error processing:

```
try
{
   // should get "ORA-942: table or view does not exist"
   stmt.execute("drop table no_such_table");
}
```

```
catch (OracleSQLException e)
{
   System.out.println(e.getMessage());
   // prints "ORA-942: table or view does not exist"

   System.out.println(e.getNumParameters());
   // prints "1"

   Object[] params = e.getParameters();
   System.out.println(params[0]);
   // prints "NO_SUCH_TABLE"
}
```

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=1521))
    (CONNECT_DATA=(SERVICE_NAME=service_name)))");
ods.setUser("scott");
ods.setPassword("tiger");
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.

In either case, 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 scott/tiger Foo1.class Foo2.class Foo3.class
  ```

- Specifying the class file names using a wildcard, as follows:

  ```
  loadjava -user scott/tiger Foo*.class
  ```

- Specifying a JAR file that contains the class files, as follows:

  ```
  loadjava -user scott/tiger Foo.jar
  ```

You can load the files using the JDBC Thin driver, as follows:

```
loadjava -thin -user scott/tiger@localhost:1521:ORCL Foo.jar
```

> **Note:** Because the server-side embedded JVM uses Java Development Kit (JDK) 1.4, it is advisable to compile classes under JDK 1.4, if they will be loaded into the server. This will catch incompatibilities during compilation, instead of at run time.

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 scott/tiger -resolve Foo.java
```

Or, use the following command to load using the JDBC Thin driver:

```
loadjava -thin -user scott/tiger@localhost:1521:ORCL -resolve Foo.java
```

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.

Server-Side Character Set Conversion of oracle.sql.CHAR Data

The server-side internal driver performs character set conversions for `oracle.sql.CHAR` in C. This is a different implementation than that for the client-side drivers, which perform character set conversions for `oracle.sql.CHAR` in Java, and offers better performance. However, `java.lang.String` offers better performance than `oracle.sql.CHAR` even in the server-side internal driver.

See Also: "Class oracle.sql.CHAR" on page 5-11

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:

- Chapter 9, "Data Sources and URLs"
- Chapter 10, "JDBC Client-Side Security Features"
- Chapter 11, "SSL Support"
- Chapter 12, "Proxy Authentication"

9

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
- Logging and Tracing

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 allows 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 file `jndi.jar` to be in the `CLASSPATH`. This file is included with the Java products on the installation CD, but is not included in the `classes12.jar` or `ojdbc14.jar` file. You must add it to the `CLASSPATH` separately. You can also obtain it from the Sun Microsystems Web site, but it is advisable to use the version from Oracle, because it has been tested with the Oracle drivers.

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 physical 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 9–1 and Table 9–2 list `OracleDataSource` properties. The properties in Table 9–1 are standard properties according to the Sun Microsystems specification. The properties in Table 9–2 are Oracle extensions.

Note: Oracle does not implement the standard `roleName` property.

Table 9–1 Standard Data Source Properties

Name	Type	Description
`databaseName`	`String`	Name of the particular database on the server. Also known as the SID in Oracle terminology.
`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

The `OracleDataSource` class implements the following setter and getter methods for the standard properties:

- `public synchronized void setDatabaseName(String dbname)`
- `public synchronized String getDatabaseName()`
- `public synchronized void setDataSourceName(String dsname)`
- `public synchronized String getDataSourceName()`
- `public synchronized void setDescription(String desc)`
- `public synchronized String getDescription()`
- `public synchronized void setNetworkProtocol(String np)`
- `public synchronized String getNetworkProtocol()`
- `public synchronized void setPassword(String pwd)`
- `public synchronized void setPortNumber(int pn)`
- `public synchronized int getPortNumber()`
- `public synchronized void setServerName(String sn)`
- `public synchronized String getServerName()`
- `public synchronized void setUser(String user)`
- `public synchronized String getUser()`

Note: For security reasons, there is no `getPassword()` method.

Table 9–2 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 the 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.
tnsEntry	String	Specifies the TNS entry name, relevant only for the OCI driver. The TNS entry name corresponds to the TNS entry specified in the tnsnames.ora configuration file. This property is only for OracleXADatasource. 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 an SQLException with error code ORA-17207 is thrown
url	String	Specifies the URL of the database connect 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.

Notes:

- 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.

The `OracleDataSource` class implements the following `set`*`XXX`* and `get`*`XXX`* methods for the Oracle extended properties:

- `String getConnectionCacheName()`
- `java.util.Properties getConnectionCacheProperties()`
- `void setConnectionCacheProperties(java.util.Properties cp)`
- `java.util.Properties getConnectionProperties()`
- `void setConnectionProperties(java.util.Properties cp)`
- `boolean getConnectionCachingEnabled()`
- `void setImplicitCachingEnabled()`
- `String getDriverType()`
- `void setDriverType(String dt)`
- `String getURL()`
- `void setURL(String url)`
- `String getTNSEntryName()`
- `void setTNSEntryName(String tns)`
- `boolean getNativeXA()`
- `void setNativeXA(boolean nativeXA)`
- `String getONSConfiguration()`
- `void setONSConfiguration(String onsConfig)`

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:scott/tiger@localhost:1521: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("dlsun999");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("816");
ods.setPortNumber(1521);
ods.setUser("scott");
ods.setPassword("tiger");
Connection conn = ods.getConnection();
```

Or, optionally, override the user name and password, as follows:

```
Connection conn = ods.getConnection("bill", "lion");
```

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("dlsun999");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("816");
ods.setPortNumber(1521);
ods.setUser("scott");
ods.setPassword("tiger");
```

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.

> **Notes:** 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

Table 9–3 provides the detailed list of connection properties that Oracle JDBC drivers support.

Table 9–3 Connection Properties Recognized by Oracle JDBC Drivers

Name	Type	Description
accumulateBatchResult	Boolean[1]	The value TRUE causes the number of modified rows used to determine when to flush a batch accumulate across all batches flushed from a single statement. The default value is FALSE, indicating each batch is counted separately.
database	String	Connect string for the database
defaultBatchValue	Integer[1]	The default batch value that triggers an execution request. The default value is 10.
defaultExecuteBatch	Integer[1]	The default batch size when using Oracle batching.
defaultNchar	Boolean[1]	The value TRUE causes the default mode for all character data columns to be NCHAR.
defaultRowPrefetch	Integer[1]	The default number of rows to prefetch from the server. The default value is 10.
disableDefineColumnType	Boolean[1]	The value TRUE causes defineColumnType to have no effect. This is highly recommended when using JDBC Thin driver, especially when the database character set contains four byte characters that expand to two UCS2 surrogate characters, for example, AL32UTF8. defineColumnType provides no benefit when used with the JDBC Thin driver in Oracle Database 10*g*. This property is provided so that you do not have to remove the calls from your code. This is especially valuable if you use the same code with the JDBC Thin driver and the JDBC OCI or server-side internal driver.
DMSName	String	Name of the Dynamic Monitoring Service (DMS) Noun that is the parent of all JDBC DMS metrics.
DMSType	String	Type of the DMS Noun that is the parent of all JDBC DMS metrics.
fixedString	Boolean[1]	The value TRUE causes JDBC to use FIXED CHAR semantics when setObject is called with a String argument. By default, JDBC uses VARCHAR semantics. The difference is in blank padding. By default, there is no blank padding. For example, 'a' does not equal 'a ' in a CHAR(4) unless fixedString is TRUE.
includeSynonyms	Boolean[1]	Set to TRUE to include column information from predefined synonym, when you run a getColumns call. This is equivalent to setIncludeSynonyms. The default value is FALSE.
internal_logon	String	The user name used in an internal logon. Must be the role, such as sysdba or sysoper, that enables you to log on as sys
oracle.jdbc.J2EE13Compliant	Boolean[1]	The value TRUE causes JDBC to use strict compliance for some edge cases. In general, Oracle JDBC drivers allow some operations that are not permitted in the strict interpretation of Java2 Platform, Enterprise Edition (J2EE) 1.3. Setting this property to TRUE will cause those cases to throw SQLException. There are some other edge cases where Oracle JDBC drivers have slightly different behavior than defined in J2EE 1.3. This results from Oracle having defined the behavior prior to the J2EE 1.3 specification and the resultant need for compatibility with existing customer code. Setting this property will result in full J2EE 1.3 compliance at the cost of incompatibility with some customer code. Can be either a system property or a connection property.

Table 9–3 (Cont.) Connection Properties Recognized by Oracle JDBC Drivers

Name	Type	Description
`oracle.jdbc.TcpNoDelay`	Boolean[1]	The value `TRUE` sets the `TCP_NODELAY` property on the socket when using the JDBC Thin driver. Can be either a system property or a connection property.
`oracle.jdbc.ocinativelibrary`	String	Name of the native library for the JDBC OCI driver. If not set, then the default name, `libocijdbc`*X*, where *X* is the version number, is used.
`password`	String	The password for logging into the database.
`processEscapes`	Boolean[1]	`TRUE` if escape processing is enabled for all statements, and `FALSE` if escape processing is disabled. The default value is `TRUE`.
`remarksReporting`	Boolean[1]	`TRUE` if `getTables` and `getColumns` should report `TABLE_REMARKS`. This is equivalent to using `setRemarksReporting`. The default value is `FALSE`.
`remarksReporting`	Boolean[1]	The value `TRUE` causes `OracleDatabaseMetaData` to include remarks in the metadata. This can result in a substantial reduction in performance.
`restrictGetTables`	Boolean[1]	The value `TRUE` causes JDBC to return a more refined value for `DatabaseMeta.getTables`. By default, JDBC will return things that are not accessible tables. These can be non-table objects or accessible synonyms for inaccessible tables. If this property is `TRUE`, then JDBC returns only accessible tables. This has a substantial performance penalty.
`server`	String	Name of the database host.
`tcp.nodelay`	Boolean[1]	Orders Oracle Net to preempt delays in buffer flushing within the TCP/IP protocol stack.
`useFetchSizeWithLongColumn`	Boolean[1]	The value `TRUE` causes JDBC to prefetch rows even when there is a `LONG` or `LONG RAW` column in the result. By default, JDBC fetches only one row at a time if there are `LONG` or `LONG RAW` columns in the result. Setting this property to `TRUE` can improve performance but can also cause `SQLException` if the results are too big. **Note:** Oracle recommends the use of `LOB` instead of `LONG` and `LONG RAW` columns.
`user`	String	The user name for logging into the database.

[1] All these entries are actually strings containing a value of the type listed.

Using Roles for SYS Login

To specify the role for the `SYS` login, use the `internal_logon` connection property. To logon as `SYS`, set the `internal_logon` connection property to `SYSDBA` or `SYSOPER`.

> **Note:** The ability to specify a role is supported only for the `sys` user name.

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 the 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/orapw password=sys entries=200
     ```

   - In Microsoft Windows

     ```
     orapwd file=%ORACLE_HOME%\database\PWDsid_name.ora password=sys entries=200
     ```

   file must be the name of the password file. password is the password for the 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.

   ```
   SQL> alter user sys identified by sys;
   ```

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 9–1 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:@inst1"; the inst1 is a remote database
String url = "jdbc:oracle:thin:@//myHost:1521/service_name";
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`.

  See Also: "Oracle Row Prefetching" on page 25-15

- Setting the `remarksReporting` property is equivalent to calling `setRemarksReporting`.

See Also: "DatabaseMetaData TABLE_REMARKS Reporting" on page 25-20

- Setting the `defaultBatchValue` property is equivalent to calling `setDefaultExecuteBatch`

See Also: "Oracle Update Batching" on page 25-3

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", "scott");
info.put ("password", "tiger");
info.put ("defaultRowPrefetch","20");
info.put ("defaultBatchValue", "5");

//specify the datasource object
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@//myhost:1521/orcl");
ods.setUser("scott");
ods.setPassword("tiger");
ods.setConnectionProperties(info);
...
```

Logging and Tracing

The data source facility offers a way to register a character stream for JDBC to use as output for error logging and tracing information. This facility enables tracing specific to a particular data source instance. If you want all data source instances to use the same character stream, then you must register the stream with each data source instance individually.

The `OracleDataSource` class implements the following standard data source methods for logging and tracing:

- `public synchronized void setLogWriter(PrintWriter pw)`
- `public synchronized PrintWriter getLogWriter()`

The `PrintWriter` class is in the standard `java.io` package.

Notes:

- When a data source instance is created, logging is disabled by default. The log stream name is initially `null`.
- Messages written to a log stream registered to a data source instance are not written to the same log stream used by `DriverManager`.
- An `OracleDataSource` instance obtained from a JNDI name lookup will not have its `PrinterWriter` set, even if the `PrintWriter` was set when a data source instance was first bound to this JNDI name.

Database URLs and Database Specifiers

Database URLs are strings. The complete URL syntax is:

`jdbc:oracle:`*`driver_type`*`:[`*`username`*`/`*`password`*`]@`*`database_specifier`*

Notes:

- 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.

Database Specifiers

Table 9–4, shows the possible database specifiers, listing which JDBC drivers support each specifier.

Notes:

- Starting Oracle Database 10*g*, Oracle Service IDs are not supported.
- Oracle Database 10*g* no longer supports Oracle Names as a naming method.

Table 9–4 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=1521))`
		`(ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521)))`
		`(CONNECT_DATA=(SERVICE_NAME=service_name)))"`
		OCI, using a cluster:
		`"jdbc:oracle:oci:@(DESCRIPTION=`
		`(ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias)`
		`(PORT=1521))`
		`(CONNECT_DATA=(SERVICE_NAME=service_name)))"`

Thin-style service name	Thin	Refer to "Thin-style Service Name Syntax" for details. `"jdbc:oracle:thin:scott/tiger@//myhost:1521/myservicename"`
LDAP syntax	Thin	Refer to LDAP Syntax for details.
Bequeath connection	OCI	Empty. That is, nothing after @ `"jdbc:oracle:oci:scott/tiger/@"`
TNSNames alias	Thin, OCI	Refer to "TNSNames Alias Syntax" for details.

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`*

Notes: *`host_name`* can be the name of a single host or a *`cluster_alias`*.

The JDBC Thin driver supports only the TCP/IP protocol.

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 `scott` with password `tiger` that has a `TNSNAMES` entry of `MyHostString`, then write the following:

```
OracleDataSource ods = new OracleDataSource();
ods.setTNSEntryName("MyTNSAlias");
ods.setUser("scott");
ods.setPassword("tiger");
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.acme.com:7777/sales,cn=OracleContext,dc=com"
```

When using SSL, change this to:

```
"jdbc:oracle:thin:@ldaps://ldap.acme.com:7777/sales,cn=OracleContext,dc=com"
```

Notes: 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.acme.com:3500/cn=salesdept,cn=OracleContext,dc=com
/salesdb " +
"ldap://ldap2.acme.com:3500/cn=salesdept,cn=OracleContext,dc=com/salesdb " +
"ldap://ldap3.acme.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 a LDAP server is configured as not allowing 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.acme.com:7777/sales,cn=salesdept,cn=OracleContext,d
c=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.

10

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), includes features to support data encryption, data integrity, third-party authentication, and authorizations. Oracle JDBC supports most of these features. However, the JDBC Thin driver must be considered separately from the JDBC OCI driver.

> **Note:** This discussion is not relevant to the server-side internal driver, given that all communication through that driver is completely internal to the server.

This chapter contains the following sections:

- Support for Oracle Advanced Security
- Support for Login Authentication
- Support for Data Encryption and Integrity
- Secure External Password Store

Support for Oracle Advanced Security

Both the JDBC OCI and the JDBC Thin drivers support at least some of the Oracle Advanced Security features. If you are using the OCI driver, then you can set relevant parameters in the same way that you would in any Oracle client setting. The JDBC Thin driver supports the Oracle Advanced Security features through a set of Java classes included with the JDBC classes in a Java Archive (JAR) file and supports security parameter settings through Java properties objects.

This section covers the following topics:

- JDBC OCI Driver Support for Oracle Advanced Security
- JDBC Thin Driver Support for Oracle Advanced Security

JDBC OCI Driver Support for Oracle Advanced Security

If you are using the JDBC OCI driver, which presumes 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.

See Also: *Oracle Database Advanced Security Administrator's Guide*

Note: The key exception to the preceding, with respect to Java, is that the Secure Sockets Layer (SSL) protocol is supported by the Oracle JDBC OCI drivers only if you use native threads in your application. This requires special attention, because green threads are generally the default.

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 your JDBC `classes12.jar` or `ojdbc14.jar` file. Security parameters for encryption and integrity, normally set in `SQLNET.ORA`, are set in a Java properties file instead.

See Also: "JDBC Thin Driver Support for Encryption and Integrity" on page 10-4

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.

The Oracle JDBC Thin driver implements Oracle O3LOGON challenge-response protocol to authenticate the user.

Note: Third-party authentication features supported by Oracle Advanced Security, such as those provided by RADIUS, Kerberos, or SecureID, are not supported by the Oracle JDBC Thin driver. For the Oracle JDBC OCI driver, support is the same as in any Oracle-client situation.

Support for Data Encryption and Integrity

You can use 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 10–1 shows how these possible settings on the client-side and server-side combine to either enable or disable the feature.

Table 10–1 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 10–1 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. And, again, the same is true for integrity.

See Also: *Oracle Database Advanced Security Administrator's Guide*

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 a 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.

To summarize, the client parameters are shown in Table 10–2:

Table 10–2 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

Note: For the Oracle Advanced Security domestic edition only, settings of RC4_128 and RC4_256 are also possible.

See Also: *Oracle Database Advanced Security Administrator's Guide*

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. Corresponding parameters can be set through a Java properties object that you would then be used when opening a database connection.

Table 10–3 lists the parameter information for the JDBC Thin driver.

Table 10–3 Thin Driver Client Parameters for Encryption and Integrity

Parameter Name	Parameter Type	Parameter Class	Possible Settings
oracle.net.encryption_client	String	static	REJECTED ACCEPTED REQUESTED REQUIRED
oracle.net.encryption_types_client	String	static	RC4_40 RC4_56 DES40C DES56C 3DES112 3DES168
oracle.net.crypto_checksum_client	String	static	REJECTED ACCEPTED REQUESTED REQUIRED
oracle.net.crypto_checksum_types_client	String	static	MD5

Notes:

- Because Oracle Advanced Security support for the Thin driver is incorporated directly into the JDBC classes JAR file, there is only one version, not separate domestic and export editions. Only parameter settings that would be 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 10–3, and then uses the properties object in opening a connection to the database:

```
...
Properties prop = new Properties();
prop.put("oracle.net.encryption_client", "REQUIRED");
prop.put("oracle.net.encryption_types_client", "( DES40 )");
prop.put("oracle.net.crypto_checksum_client", "REQUESTED");
prop.put("oracle.net.crypto_checksum_types_client", "( MD5 )");

OracleDataSource ods = new OracleDataSource();
ods.setProperties(prop);
ods.setURL("jdbc:oracle:thin:@localhost:1521:main");
Connection conn = ods.getConnection();
...
```

The parentheses around the parameter values in the `encryption_types_client` and `crypto_checksum_types_client` settings allow for lists of values. Currently, the JDBC Thin driver supports only one possible value in each case. However, in the future, when multiple values are supported, specifying a list will result in a negotiation between the server and the client that determines which value is actually used.

Example

Example 10–1 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 hardcoding them as in the previous examples

Before running this example, you must turn on encryption in the `sqlnet.ora` file. For example, the following four lines will turn on DES40, DES, 2-DES-112 and 3-DES168 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= (DES40, DES, 3DES112, 3DES168)
SQLNET.CRYPTO_SEED = 12345678901234567890
```

Example 10–1 Setting Data Encryption and Integrity Parameters

```
import java.sql.*;
import java.sql.*;
import java.io.*;
import java.util.*;
import oracle.net.ns.*;
import oracle.net.ano.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;

class Employee
{
  public static void main (String args [])
    throws Exception
  {

    Properties props = new Properties();

    try {
      FileInputStream defaultStream = new FileInputStream(args[0]);
      props.load(defaultStream);


      int level = AnoServices.REQUIRED;
      props.put("oracle.net.encryption_client", Service.getLevelString(level));
      props.put("oracle.net.encryption_types_client", "( 3DES168 )");
      props.put("oracle.net.crypto_checksum_client",
                Service.getLevelString(level));
      props.put("oracle.net.crypto_checksum_types_client", "( MD5 )");
    } catch (Exception e) { e.printStackTrace(); }


    // You can put a database name after the @ sign in the connection URL.
    OracleDataSource ods = new OracleDataSource();
    ods.setURL("jdbc:oracle:thin:@//host.example.com:1521/main.example.com");
    ods.setConnectionProperties(props);
    Connection conn = ods.getConnection();


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

    // Select the ENAME column from the EMP table
    ResultSet rset = stmt.executeQuery ("select ENAME from EMP");

    // Iterate through the result and print the employee names
    while (rset.next ())
       System.out.println (rset.getString (1));

    conn.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 clear in scripts and application code, and simplifies maintenance because you need not change your code each time user names and passwords change. In addition, not having to change application code also makes it easier to enforce password management policies for these user accounts.

When you configure a client to use the external password store, applications can use the following syntax to connect to databases that use password authentication:

```
CONNECT /@database_alias
```

Note that you need not specify database login credentials in this `CONNECT` statement. Instead your system looks for database login credentials in the client wallet.

> **See Also:** *Oracle Database Security Guide* for information about configuring your client to use secure external password store and for information about managing credentials in it

11

SSL Support

The Secure Sockets Layer (SSL) protocol and the Transport Layer Security (TLS) protocol help protect the privacy and integrity of data that is transported over a network, such as the Internet. This chapter provides an overview of the SSL protocol and discusses its support in the JDBC Thin driver.

This chapter consists of the following sections:

- Overview of SSL
- Java Version of SSL
- SSL in JDBC

Overview of SSL

SSL is a widely used protocol that provides secure communication over a network. It uses several different cryptographic processes to ensure that the data sent over the network is secure. It provides a secure enhancement to the standard TCP/IP protocol, which is used for Internet communication. SSL uses public key cryptography to provide authentication. SSL also uses secret key cryptography and digital signatures to ensure privacy and integrity of data.

Data communication between a client and a server over an SSL enabled network begins with an exchange of information called SSL handshake. The SSL session first decides on a cipher suite, which is a set of cryptographic algorithms and key sizes, that will be used by the client and the server. Optionally, the client then authenticates the server to ensure that the server represents the entity with which the client wants to interact. The server too can authenticate the client before exchanging data. After authenticating, the client and server exchange data. The data is encrypted using the already agreed upon cryptographic hash function.

Java Version of SSL

The Java Secure Socket Extension (JSSE) enables secure Internet communication by providing a framework and an implementation for a Java version of the SSL and TLS protocols. It provides support for data encryption, server authentication, optional client authentication, and message integrity. JSSE 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 supports SSL version 2.0 and 3.0.

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 sockets, server sockets, SSL sockets, and SSL server sockets. These classes also enable you to configure the 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.

> **See Also:** For more information about JSSE and its APIs, refer to *Java Secure Socket Extension (JSSE) Reference Guide* for J2SDK 1.4.2.

SSL in JDBC

Oracle Database 10*g* release 2 (10.2) implements JSSE to provide secure communication between client and server. The JDBC Thin driver makes use of the JSSE APIs, abstracting the complexity of creating SSL enabled connections. The application developers are not exposed to the overhead involved in using the JSSE APIs.

> **Note:** SSL support is only available with JDK 1.4 and later.

Oracle Database 10g provides the following sets of properties that can be used to create SSL enabled connection between client and server:

- Non-JSSE Related Properties
- JSSE Related Properties
- Enabling SSL

> **Note:** A client can choose to establish either an SSL connection or a non-SSL connection.

> **See Also:** *Oracle Database Advanced Security Administrator's Guide*

Non-JSSE Related Properties

Oracle Database 10g defines certain properties, which are non-JSSE specific, to configure an SSL connection. These properties are available with both the JDBC OCI and Thin drivers. The following properties are defined:

- `oracle.net.ssl_version`

  This property is used to set the SSL version to be used by the JDBC driver. It takes a `String` value. The value specified should be supported by both the SSL implementation and the server. The default value of the property is `ANY`. In the case of the default value, first TLS 1.0 is tried followed by SSL 3.0 and then SSL 2.0.

- `oracle.net.wallet_location`

  This property is used to specify the location of the wallet. The wallet contains the certificates and keys used by SSL. This property takes a `String` value that identifies a valid location of the wallet. It does not have any default value.

Note: If this property is set, then it overrides the value of the `javax.net.ssl.keyStore` JSSE property and the following values will be used for other JSSE properties:

```
javax.net.ssl.keyStorePassword = null
javax.net.ssl.keyStoreType = SSO
```

The value of `javax.net.ssl.trustStore` will be same as that of `javax.net.ssl.keyStore`.

Only `METHOD=file` is supported in Oracle Database 10*g* release 2 (10.2).

- `oracle.net.ssl_cipher_suites`

  This property is used to specify a cipher suite that will be used by the SSL connection. It takes a `String` value that specifies the algorithms and the key sizes for data integrity and encryption, which are separated by commas. The cipher suite specified should be supported by both the SSL implementation and the server. This property does not have any default value.

- `oracle.net.ssl_server_dn_match`

  This property is used to force the distinguished name (dn) of the server to match with its service name. It takes a `String` value. The permitted values are `TRUE`, `FALSE`, `YES`, `NO`, `ON`, and `OFF`. The values `TRUE`, `YES`, and `ON` force the dn of the server to match with its service name. The default value is `FALSE`.

Note: The property values are not case-sensitive. The properties are set as key/value pairs using a `java.util.Properties` object.

Example

A sample code illustrating the use of the non-JSSE properties is as follows:

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

//specify the properties object
java.util.Properties info = new java.util.Properties();
...
// Set the SSL version
info.put ("oracle.net.ssl_version","3.0");

// Set the wallet location
info.put ("oracle.net.wallet_location",
"(SOURCE=(METHOD=file)(METHOD_DATA=(DIRECTORY=directory)))");

// Set the cipher suite
info.put("oracle.net.ssl_cipher_suites","SSL_DH_DSS_WITH_DES_CBC_SHA");

// Force dn to match service name
info.put("oracle.net.ssl_serevr_dn_match","TRUE");
...

//specify the datasource object
```

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@//myhost:1521/orcl");
ods.setUser("scott");
ods.setPassword("tiger");
ods.setConnectionProperties(info);
...
```

JSSE Related Properties

Oracle Database 10g supports certain properties that are defined by the JSSE framework for configuring SSL. These properties are available only with the JDBC Thin driver. Any of these properties can be set on a per-connection basis. When a value is set for any of these properties, the corresponding system or security property is overridden. The following properties are defined:

- `javax.net.ssl.keyStore`

  This property is used to specify the location of the key store. A key store is a database of key material that are used for various purposes, including authentication and data integrity. This property takes a `String` value. Any valid key store location can be assigned to this property. The default value is `NULL`.

- `javax.net.ssl.keyStoreType`

  This property denotes the type of the key store. It takes a `String` value. Any valid key store type supported by SSL can be assigned to this property. The default value is `KeyStore.getDefaultValue()`.

- `javax.net.ssl.keyStorePassword`

  This property specifies the password of the key store. This password value is used to check the integrity of the data in the key store before accessing it. This property takes a `String` value. It does not have a default value.

- `javax.net.ssl.trustStore`

  This property is used to specify the location of the trust store. A trust store is a key store that is used when making decisions about which clients and servers can be trusted. The property takes a `String` value that specifies a valid trust store location. The default value is `jssecacerts`, if available, or `cacerts`.

- `javax.net.ssl.trustStoreType`

  This property denotes the type of the trust store. It takes a `String` value. Any valid trust store type supported by SSL can be assigned to this property. The default value is `TrustStore.getDefaultValue()`.

- `javax.net.ssl.trustStorePassword`

  This property is used to set the password for the trust store. The trust store password is used to check the integrity of the data in the trust store before accessing it. The property takes a `String` value. It does not have any default value.

Example

A sample code illustrating the use of the JSSE properties is as follows:

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

```
//specify the properties object
java.util.Properties info = new java.util.Properties();
...
// Set the key store, type, and password
info.put ("javax.net.ssl.keyStore","file_location");
info.put ("javax.net.ssl.keyStoreType","JKS");
info.put ("javax.net.ssl.keyStorePassword","password");

// Set the trust store, type, and password
info.put("javax.net.ssl.trustStore","location");
info.put("javax.net.ssl.trustStoreType","JKS");
info.put("javax.net.ssl.trustStorePassword","password");
...

//specify the datasource object
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@//myhost:1521/orcl");
ods.setUser("scott");
ods.setPassword("tiger");
ods.setConnectionProperties(info);
...
```

Enabling SSL

An Oracle JDBC user has to specify `tcps` and a DN. The following is an example:

```
(DESCRIPTION=
  (ADDRESS_LIST=
    (ADDRESS= (PROTOCOL = tcps) (HOST = finance_server) (PORT = 1521)))
(CONNECT_DATA=
  (SERVICE_NAME= Finance.us.acme.com))
(SECURITY=
  (SSL_SERVER_CERT_DN="cn=finance,cn=OracleContext,c=us,o=acme"))
```

See Also: *Oracle Database Advanced Security Administrator's Guide* and *Oracle Database Security Guide*

Note: The JDBC Thin driver requires the user to authenticate by providing a user name and password even if the SSL authentication has happened successfully. The JDBC Thin driver does not use the authentication part of the SSL handshake.

12

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:

- Need for Proxy Authentication
- Creating Proxy Connections
- Caching Proxy Connections

Need for Proxy Authentication

Proxy authentication allows one JDBC connection to act as a proxy for other JDBC connections. An application may need proxy authentication for any of the following reasons:

- The middle tier does not know the password of the proxy user.

  It is sometimes a security concern for the middle tier to know the passwords of all the database users. In this case, proxy authentication is done by first issuing an `ALTER USER` statement. For example:

  ```
  ALTER USER jeff GRANT CONNECT THROUGH scott WITH ROLES role1, role2;
  ```

  After the `ALTER USER` statement is processed, the application can connect as `jeff` using the already authenticated credentials of `scott`. Although the created session will function as if `jeff` was connected normally, `jeff` will not have to divulge its password to the middle tier. The proxy section has access to the schema of `jeff`, as well as to what is indicated in the list of roles. Therefore, if `scott` wants `jeff` to access its `EMP` table, then the following code can be used:

  ```
  CREATE ROLE role1;
  GRANT SELECT ON emp TO role1;
  ```

  The role clause limits the user `jeff` to access only those database objects of `scott` that are mentioned in the list of the roles. The list of roles can be empty.

- Accounting purposes.

  The transactions made through proxy sessions can be better accounted by proxying the connecting user, `jeff`, under different users, such as `scott` and `scott2`, assuming `scott` and `scott2` are authenticated. Transactions made under these different proxy sessions by `jeff` can be logged separately.

Note: In this chapter, a JDBC connection to a database is a user session in the database and vice versa.

Creating Proxy Connections

In order to get a proxy connection, a user, say `jeff`, has to connect to the database through another user, say `scott`. The proxy user, `scott`, should have an active authenticated connection. Using this active connection, the driver sends issues 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.

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 scott AUTHENTICATED USING password;
  ```

  In this case, `jeff` is the user name and `scott` is the proxy for `jeff`.

  The reason why the password option exists is 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 scott
  ```

- `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 scott AUTHENTICATED USING DISTINGUISHED
  NAME;
  ```

- `CERTIFICATE`

  This is a more encrypted way of passing the credentials of the user, which 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 scott AUTHENTICATED USING CERTIFICATE;
  .
  ```

Note: The use of certificates for proxy authentication will be desupported in future Oracle Database releases.

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, `scott`, is not available until the new session, `jeff`, is closed.

Notes:

- All the options can be associated with roles.
- When opening a new proxied connection, a new session is started on the database server. Along with this session a new local transaction is created.

Caching Proxy Connections

Proxy connections, like normal connections, can be cached. Caching proxy connections enhances the performance. To create a proxy connection, you need to first create a connection using one of the `getConnection` methods on a cache enabled `OracleDataSource` object. A proxy session is then created by calling the `openProxySession` method on the obtained connection.

See Also: Chapter 23, "Implicit Connection Caching"

When the `close(PROXY_CONNECTION)` method is called on this physical connection, the proxy session created on this connection gets closed. 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.

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. Implicit connection cache supports caching of any user/password authenticated connection. Therefore, any user authenticated proxy connection can be cached and retrieved.

A sample code illustrating the use of proxy sessions is as follows:

```
...
java.util.Properties connAttr = null;
...
//obtain connection from a cache enabled DataSource
OracleConnection conn = ds.getConnection("scott",tiger");
conn.openProxySession(proxyType, proxyProps);
...
connAttr.setProperty("CONNECTION_TAG","JOE'S_PROXY_CONNECTION");
conn.applyConnectionAttributes(connAttr); //apply attributes to the connection
conn.close(); //return the tagged connection to the cache
...
//come back later and ask for Joe's proxy connection
conn=ds.getConnection(connAttr); //This will retrieve Joe's proxy connection
...
```

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. This can be done by calling the `close(INVALID_CONNECTION)` method on the connection. The method closes both the connection and the proxy session and removes them from the cache.

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 on 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:

- Chapter 13, "Accessing and Manipulating Oracle Data"
- Chapter 14, "Java Streams in JDBC"
- Chapter 15, "Working with Oracle Object Types"
- Chapter 16, "Working with LOBs and BFILEs"
- Chapter 17, "Using Oracle Object References"
- Chapter 18, "Working with Oracle Collections"
- Chapter 19, "Result Set"
- Chapter 20, "JDBC RowSets"
- Chapter 21, "Globalization Support"

13

Accessing and Manipulating Oracle Data

This chapter describes data access in `oracle.sql.*` formats, as opposed to standard Java formats. The `oracle.sql.*` formats are a key factor of the Oracle Java Database Connectivity (JDBC) extensions, offering significant advantages in efficiency and precision in manipulating SQL data.

Using `oracle.sql.*` formats 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 Meta Data Extensions

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 13–1 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 10*g*. 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 a `oracle.sql.*` Java type. Manipulating SQL data as `oracle.sql.*` data types minimizes conversions, improving performance and eliminating conversion losses.

See Also: "Package oracle.sql" on page 5-4

Table 13–1 Default Mappings Between SQL Types and Java Types

SQL Data Types	JDBC Type Codes	Standard Java Types	Oracle Extension Java Types
	STANDARD JDBC 1.0 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
	STANDARD JDBC 2.0 TYPES:		
BLOB	java.sql.Types.BLOB	java.sql.Blob	oracle.sql.BLOB
CLOB	java.sql.Types.CLOB	java.sql.Clob	oracle.sql.CLOB
user-defined object	java.sql.Types.STRUCT	java.sql.Struct	oracle.sql.STRUCT
user-defined reference	java.sql.Types.REF	java.sql.Ref	oracle.sql.REF
user-defined collection	java.sql.Types.ARRAY	java.sql.Array	oracle.sql.ARRAY
	ORACLE EXTENSIONS:		
BFILE	oracle.jdbc.OracleTypes.BFILE	NA	oracle.sql.BFILE

Table 13–1 (Cont.) Default Mappings Between SQL Types and Java Types

SQL Data Types	JDBC Type Codes	Standard Java Types	Oracle Extension Java Types
ROWID	oracle.jdbc.OracleTypes.ROWID	NA	oracle.sql.ROWID
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

Note: For database versions, such as 8.1.7, which do not support the TIMESTAMP data type, TIMESTAMP is mapped to DATE.

See Also :

- "Valid SQL-JDBC Data Type Mappings" on page A-1
- Chapter 5, "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.sql.ORAData

See Also: "Mapping Oracle Objects" on page 15-1 and "Creating and Using Custom Object Classes for Oracle Objects" on page 15-8

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. The Oracle Database 10*g* 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.sql.OraData` rather than the `java.sql.SqlData` if the OraData 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.DATE` or `oracle.sql.TIMESTAMP` rather than `java.sql.Date` or `java.sql.Timestamp` if you are using JDK 5.0 or earlier versions or require maximum performance. You may also use the `oracle.sql` data type if you want to read many date values or compute or display only a small percentage. Due to a bug in all versions of Java prior to JDK 5.1, construction of `java.lang.Date` and `java.lang.Timestamp` objects is slow, especially in multithreaded environments. This bug is fixed in JDK 5.1.

> **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 wrapper type corresponding to every primitive type that can represent `null`. The object wrapper 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 `COMM` column contains one or more `NULL` values.

```
PreparedStatement pstmt = conn.prepareStatement(
  "SELECT * FROM EMP WHERE COMM = ?");
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 ENAMES from the EMP table that are NULL, if there is no value of 100 for COMM.

```
PreparedStatement pstmt = conn.prepareStatement("SELECT ENAME FROM EMP
  WHERE COMM =? OR  ((COMM IS NULL) AND (? IS NULL))");
pstmt.setBigDecimal(1, new BigDecimal(100));
pstmt.setNull(2, java.sql.Types.VARCHAR);
```

Result Set and Statement Extensions

The JDBC `Statement` object returns an `OracleResultSet` object, typed as 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 `ResultSet` 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 emp");
```

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.

Similarly, when you use `executeQuery` to run a stored procedure using a callable statement, the returned object is an `OracleCallableStatement`. The type of the return value of `executeQuery()` is `java.sql.CallableStatement`. If your application needs only the standard JDBC methods, you need not cast the variable. However, to take advantage of the Oracle extensions, you must cast the variable to `OracleCallableStatement`. Similar rules apply to `prepareStatement`, `prepareCall`, and so on.

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.

Although there are specific `get`*`XXX`* methods for all the Oracle SQL types, you can use the general `get` methods for convenience or simplicity, or if you are not certain in advance what type of data you will receive.

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 emp.deptno, dept.deptno
FROM emp, dept");
rset.getInt("emp.deptno");
```

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, `getObject` 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.STRUCT` 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 a `ResultSet`, you must cast the result set itself 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 ResultSet

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 "SCOTT" to the prepared call and registers the DATE type as an output parameter. After the call is run, getOracleObject retrieves the date associated with "SCOTT". 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, "SCOTT");
cstmt.registerOutParameter (2, Types.DATE);
cstmt.execute ();

DATE date = (DATE) ((OracleCallableStatement)cstmt).getOracleObject (2);
...
```

Summary of getObject and getOracleObject Return Types

Table 13–2 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:

- getObjectalways 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 13–2 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
LONG	String	oracle.sql.CHAR

Table 13–2 (Cont.) getObject and getOracleObject Return Types

Oracle SQL Type	getObject Underlying Return Type	getOracleObject Underlying Return Type
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.sql.BLOB	oracle.sql.BLOB
CLOB	oracle.sql.CLOB	oracle.sql.CLOB
BFILE	oracle.sql.BFILE	oracle.sql.BFILE
Oracle object	class specified in type map or oracle.sql.STRUCT (if no type map entry)	oracle.sql.STRUCT
Oracle object reference	oracle.sql.REF	oracle.sql.REF
collection (varray or nested table)	oracle.sql.ARRAY	oracle.sql.ARRAY

[1] ResultSet.getObject returns java.sql.Timestamp only if the oracle.jdbc.J2EE13Compliant connection property is set to TURE, else the method returns oracle.sql.TIMESTAMP.

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 `classes12dms.jar` and `ojdbc14dms.jar` files in `CLASSPATH`. These files set `oracle.jdbc.J2EE13Compliant` to `TRUE` by default. These files 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 behaviour 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 on 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` classes 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

Table 13–3 summarizes the return types for each `get`*`XXX`* method and specifies which are Oracle extensions under Java Development Kit (JDK) 1.2.x. You must cast the returned object to `OracleResultSet` or `OracleCallableStatement` to use methods that are Oracle extensions.

Table 13–3 Summary of getXXX Return Types

Method	Return Type (type in method signature)	Type of returned object	Oracle Ext for JDK 1.2.x?
getArray	java.sql.Array	oracle.sql.ARRAY	No
getARRAY	oracle.sql.ARRAY	oracle.sql.ARRAY	Yes
getAsciiStream	java.io.InputStream	java.io.InputStream	No
getBfile	oracle.sql.BFILE	oracle.sql.BFILE	Yes
getBFILE	oracle.sql.BFILE	oracle.sql.BFILE	Yes
getBigDecimal (see Notes)	java.math.BigDecimal	java.math.BigDecimal	No
getBinaryStream	java.io.InputStream	java.io.InputStream	No
getBlob	java.sql.Blob	oracle.sql.BLOB	No
getBLOB	oracle.sql.BLOB	oracle.sql.BLOB	Yes
getBoolean (see Notes)	boolean	boolean	No
getByte	byte	byte	No
getBytes	byte[]	byte[]	No
getCHAR	oracle.sql.CHAR	oracle.sql.CHAR	Yes
getCharacterStream	java.io.Reader	java.io.Reader	No
getClob	java.sql.Clob	oracle.sql.CLOB	No
getCLOB	oracle.sql.CLOB	oracle.sql.CLOB	Yes
getDate	java.sql.Date	java.sql.Date	No
getDATE	oracle.sql.DATE	oracle.sql.DATE	Yes
getDouble	double	double	No
getFloat	float	float	No
getInt	int	int	No
getINTERVALDS	oracle.sql.INTERVALDS	oracle.sql.INTERVALDS	Yes
getINTERVALYM	oracle.sql.INTERVALYM	oracle.sql.INTERVALYM	Yes
getLong	long	long	No
getNUMBER	oracle.sql.NUMBER	oracle.sql.NUMBER	Yes
getOracleObject	oracle.sql.Datum	subclasses of oracle.sql.Datum	Yes
getRAW	oracle.sql.RAW	oracle.sql.RAW	Yes
getRef	java.sql.Ref	oracle.sql.REF	No
getREF	oracle.sql.REF	oracle.sql.REF	Yes
getROWID	oracle.sql.ROWID	oracle.sql.ROWID	Yes
getShort	short	short	No
getString	String	String	No

Table 13–3 (Cont.) Summary of getXXX Return Types

Method	Return Type (type in method signature)	Type of returned object	Oracle Ext for JDK 1.2.x?
getSTRUCT	oracle.sql.STRUCT	oracle.sql.STRUCT.	Yes
getTime	java.sql.Time	java.sql.Time	No
getTimestamp	java.sql.Timestamp	java.sql.Timestamp	No
getTIMESTAMP	oracle.sql.TIMESTAMP	oracle.sql.TIMESTAMP	Yes
getTIMESTAMPTZ	oracle.sql.TIMESTAMPTZ	oracle.sql.TIMESTAMPTZ	Yes
getTIMESTAMPLTZ	oracle.sql.TIMESTAMPLTZ	oracle.sql.TIMESTAMPLTZ	Yes
getUnicodeStream	java.io.InputStream	java.io.InputStream	No
getURL	java.net.URL	java.net.URL	No

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 normally 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);
```

Alternatively, you can return the object into a generic `oracle.sql.Datum` return variable and cast it later when you use `NUMBER`-specific methods.

```
Datum rawdatum = ors.getOracleObject(1);
...
CharacterSet cs = ((NUMBER) rawdatum).FIXME();
```

This uses the `FIXME` method of `oracle.sql.NUMBER`. The `FIXME` method is not defined on `oracle.sql.Datum` and would not be reachable without the cast.

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 Parameter Types of setXXX Methods
- Setter Method Size Limitations
- Setter Methods That Take Additional Input
- Method setFixedCHAR for Binding CHAR Data into WHERE Clauses

Input Parameter Types of setXXX Methods

Table 13–4 summarizes the input types for all the `setXXX` methods and specifies which are Oracle extensions under JDK 1.2.x. To use methods that are Oracle extensions, you must cast your statement to `OraclePreparedStatement` or `OracleCallableStatement`.

> **See Also:** Table A–1, " Valid SQL Data Type-Java Class Mappings" on page A-1

Table 13–4 Summary of setXXX Input Parameter Types

Method	Input Parameter Type	Oracle Ext for JDK 1.2.x?
`setArray`	`java.sql.Array`	No
`setARRAY`	`oracle.sql.ARRAY`	Yes

Table 13–4 (Cont.) Summary of setXXX Input Parameter Types

Method	Input Parameter Type	Oracle Ext for JDK 1.2.x?
setAsciiStream (see Notes section)	java.io.InputStream	No
setBfile	oracle.sql.BFILE	Yes
setBFILE	oracle.sql.BFILE	Yes
setBigDecimal	BigDecimal	No
setBinaryFloat	float or oracle.sql.BINARY_FLOAT	Yes
setBinaryDouble	double or oracle.sql.BINARY_DOUBLE	Yes
setBinaryStream (see Notes section)	java.io.InputStream	No
setBlob	java.sql.Blob	No
setBLOB	oracle.sql.BLOB	Yes
setBoolean	boolean	No
setByte	byte	No
setBytes	byte[]	No
setCHAR (also see setFixedCHAR method)	oracle.sql.CHAR	Yes
setCharacterStream (see Notes section)	java.io.Reader	No
setClob	java.sql.Clob	No
setCLOB	oracle.sql.CLOB	Yes
setDate (see Notes section)	java.sql.Date	No
setDATE	oracle.sql.DATE	Yes
setDouble	double	No
setFixedCHAR (see setFixedCHAR section)	java.lang.String	Yes
setFloat	float	No
setInt	int	No
setINTERVALDS	oracle.sql.INTERVALDS	Yes
setINTERVALYM	oracle.sql.INTERVALYM	Yes
setLong	long	No
setNUMBER	oracle.sql.NUMBER	Yes
setRAW	oracle.sql.RAW	Yes
setRef	java.sql.Ref	No

Table 13–4 (Cont.) Summary of setXXX Input Parameter Types

Method	Input Parameter Type	Oracle Ext for JDK 1.2.x?
setREF	oracle.sql.REF	Yes
setROWID	oracle.sql.ROWID	Yes
setShort	short	No
setString	String	No
setSTRUCT	oracle.sql.STRUCT	Yes
setTime	java.sql.Time	No
setTimestamp	java.sql.Timestamp	No
setTIMESTAMP	oracle.sql.TIMESTAMP	Yes
setTIMESTAMPTZ	oracle.sql.TIMESTAMPTZ	Yes
setTIMESTAMPLTZ	oracle.sql.TIMESTAMPLTZ	Yes
setUnicodeStream	java.io.InputStream	No
setURL	java.net.URL	No

Setter Method Size Limitations

Table 13–5 lists size limitations for the `setBytes` and `setString` methods for SQL binds.

Note: These limitations do not apply to PL/SQL binds.

Table 13–5 Size Limitations for setBytes and setString Methods

Method Name	Size Limit
setBytes	2000 bytes
setString	4000 bytes

See Also: "Using Streams to Avoid Limits on setBytes and setString" on page 14-9, for information about how to work around these limits using the stream application programming interface (API).

Setter Methods That Take Additional Input

The following `set`*XXX* methods take an additional input parameter other than the parameter index and the data item itself:

- `setAsciiStream(int paramIndex, InputStream istream, int length)`

  Takes the length of the stream, in bytes.

- `setBinaryStream(int paramIndex, InputStream istream, int length)`

  Takes the length of the stream, in bytes.

- `setCharacterStream(int paramIndex, Reader reader, int length)`

  Takes the length of the stream, in characters.

- `setUnicodeStream(int paramIndex, InputStream istream, int length)`

  Takes the length of the stream, in bytes.

The particular usefulness of the `setCharacterStream` method is that when a very large Unicode value is input to a `LONGVARCHAR` parameter, it can be more practical to send it through a `java.io.Reader` object. JDBC will read the data from the stream as needed, until it reaches the end-of-file mark. The JDBC driver will do any necessary conversion from Unicode to the database character format.

Note: The preceding stream methods can also be used for large objects (LOBs). Refer to "Reading and Writing BLOB and CLOB Data" on page 16-4 for more information.

Some of the other methods that take an additional parameter other than the parameter index and the data item itself are:

- `setDate(int paramIndex, Date x, Calendar cal)`
- `setTime(int paramIndex, Time x, Calendar cal)`
- `setTimestamp(int paramIndex, Timestamp x, Calendar cal)`

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 Meta Data Extensions

The `oracle.jdbc.OracleResultSetMetaData` interface is JDBC 2.0-compliant but does not implement the `getSchemaName` and `getTableName` methods because Oracle Database 10*g* does not make this feasible. Oracle does implement many methods to retrieve information about an Oracle result set, however.

Key methods include the following:

- `int getColumnCount()`

  Returns the number of columns in an Oracle result set

- `String getColumnName(int column)`

  Returns the name of a specified column in an Oracle result set

- `int getColumnType(int column)`

  Returns the SQL type of a specified column in an Oracle result set. If the column stores an Oracle object or collection, then this method returns `OracleTypes.STRUCT` or `OracleTypes.ARRAY` respectively.

- `String getColumnTypeName(int column)`

  Returns the SQL type name for a specified column of type `REF`, `STRUCT`, or `ARRAY`. If the column stores an array or collection, then this method returns its SQL type name. If the column stores `REF` data, then this method returns the SQL type name of the objects to which the object reference points.

The following example uses several of the methods in the `OracleResultSetMetadata` interface to retrieve the number of columns from the `EMP` table and the numerical type and SQL type name of each column:

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rset = dbmd.getTables("", "SCOTT", "EMP", 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
```

14

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). Methods associated with streams let you read the data incrementally.

This chapter 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
- Streaming and Row Prefetching
- Closing a Stream
- Notes and Precautions on Streams

Overview

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.

See Also: Chapter 16, "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. After a call to executeQuery or next, the data of the LONG column is waiting to be read.

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.

Tip: Chapter 21, "Globalization Support" and "Data Streaming and Multiple Columns" on page 14-7

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.

Table 14–1 summarizes LONG and LONG RAW data conversions for each stream type.

Table 14–1 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 get*XXX*Stream 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', '0001020304050607080 9');
```

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

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 and KPRB drivers 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

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 `get`*`XXX`*`Stream` 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

- 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 necessary bits of data move across the network. By contrast, when streaming a `LONG` or `LONG RAW`, all the data always moves across the network.

Streaming BLOBs and CLOBs

When a query fetches one or more `CLOB` or `BLOB` columns, the JDBC driver transfers the data to the client. This data can be accessed as a stream. To manipulate CLOB or BLOB data from JDBC, use methods in the Oracle extension classes `oracle.sql.BLOB` and `oracle.sql.CLOB`. These classes provide specific functionality, such as reading from the CLOB or BLOB into an input stream, writing from an output stream into a CLOB or BLOB, determining the length of a CLOB or BLOB, and closing a CLOB or BLOB.

> **See Also:** "Reading and Writing BLOB and CLOB Data" on page 16-4 and "Data Interface for LOBs" on page 16-12

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.

> **See Also:** "Reading BFILE Data" on page 16-18

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 14-8

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.

Tip: "Streaming LOBs and External Files" on page 14-6

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. For `LONG` and `LONG RAW` columns, you can also close and discard the stream by closing the corresponding result set or connection object.

> **See Also:** "Bypassing Streaming Data Columns" on page 14-7 and "Streaming Data Precautions" on page 14-8

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

In Oracle Database 10*g*, the size limitation on data that may be used with `setBytes` and `setString` have been reduced and, in certain cases, eliminated. 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 16-12 and release notes for details

Streaming and Row Prefetching

If the JDBC driver encounters a column containing a data stream, then row prefetching is set back to one. Row prefetching is an Oracle performance enhancement that allows multiple rows of data to be retrieved with each trip to the database.

> **See Also:** "Oracle Row Prefetching" on page 25-15

15

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 `ORAData` interface.

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 Application Developer's Guide - Object-Relational Features*

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.sql.ORAData` 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 will materialize the object as an instance of the `oracle.sql.STRUCT` class.

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:

- STRUCT Class Functionality
- Retrieving STRUCT Objects and Attributes
- Creating STRUCT Objects and Descriptors
- Binding STRUCT Objects into Statements
- STRUCT Automatic Attribute Buffering

STRUCT Class Functionality

This section discusses standard versus Oracle-specific features of the `oracle.sql.STRUCT` class, introduces `STRUCT` descriptors, and lists methods of the `STRUCT` class to give an overview of its functionality.

Standard java.sql.Struct Methods

If your code must comply with standard JDBC 2.0, then use a `java.sql.Struct` instance and use the following standard methods:

- `getAttributes(map)`

This method retrieves the values of the attributes, using entries in the specified type map to determine the Java classes to use in materializing any attribute that is a structured object type. The Java types for other attribute values would be the same as for a `getObject` call on data of the underlying SQL type.

- `getAttributes`

  This method is the same as the preceding `getAttributes(map)` method, except it uses the default type map for the connection.

- `getSQLTypeName`

  This method returns a Java `String` that represents the fully qualified name of the Oracle object type that this `Struct` represents.

Oracle oracle.sql.STRUCT Class Methods

If you want to take advantage of the extended functionality offered by Oracle-defined methods, then use an `oracle.sql.STRUCT` instance.

The `oracle.sql.STRUCT` class implements the `java.sql.Struct` interface and provides extended functionality beyond the JDBC 2.0 standard.

The `STRUCT` class includes the following methods in addition to standard `Struct` functionality:

- `getOracleAttributes`

  Retrieves the values of the values array as `oracle.sql.*` objects

- `getDescriptor`

  Returns the `StructDescriptor` object for the SQL type that corresponds to this `STRUCT` object

- `getJavaSQLConnection`

  Returns the current connection instance

- `toJdbc`

  Consults the default type map of the connection to determine what class to map to and, then, uses `toClass`

- `toJdbc(map)`

  Consults the specified type map to determine what class to map to, and then uses `toClass`

STRUCT Descriptors

Creating and using a `STRUCT` object requires a descriptor, which is an instance of the `oracle.sql.StructDescriptor` class, to exist for the SQL type that will correspond to the `STRUCT` object. You need only one `StructDescriptor` object for any number of `STRUCT` objects that correspond to the same SQL type.

If your application is fetching objects from the database as `STRUCT`, then the Oracle JDBC drivers will construct the necessary `STRUCT` descriptors for you. You only need to construct your own `STRUCT` descriptors if you are creating `STRUCT` values yourself for use in a `setObject` call.

See Also: "Creating STRUCT Objects and Descriptors" on page 15-5

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 normally 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 an oracle.sql.STRUCT Object

You can retrieve an Oracle object directly into an `oracle.sql.STRUCT` instance. In the following example, `getObject` is used to get a `type_struct` object from the `col1` column of the table `struct_table`. Because `getObject` returns an `Object` type, the return is cast to `oracle.sql.STRUCT`. This example assumes that the `Statement` object `stmt` has already been created.

```
String cmd;
cmd = "CREATE TYPE type_struct AS object (field1 NUMBER,field2 DATE)";
stmt.execute(cmd);

cmd = "CREATE TABLE struct_table (col1 type_struct)";
stmt.execute(cmd);

cmd = "INSERT INTO struct_table VALUES (type_struct(10,'01-apr-01'))";
stmt.execute(cmd);

cmd = "INSERT INTO struct_table VALUES (type_struct(20,'02-may-02'))";
stmt.execute(cmd);

ResultSet rs= stmt.executeQuery("SELECT * FROM struct_table");
oracle.sql.STRUCT oracleSTRUCT=(oracle.sql.STRUCT)rs.getObject(1);
```

Another way to return the object as a `STRUCT` object is to cast the result set to `OracleResultSet` and use the Oracle extension `getSTRUCT` method:

```
oracle.sql.STRUCT oracleSTRUCT=((OracleResultSet)rs).getSTRUCT(1);
```

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`. Because `getObject` returns a `java.lang.Object`, 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:** The 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`.

Creating STRUCT Objects and Descriptors

This section describes how to create `STRUCT` objects and descriptors and lists useful methods of the `StructDescriptor` class.

> **Note:** If you have already fetched from the database a `STRUCT` of the appropriate SQL object type, then the easiest way to get a `STRUCT` descriptor is to call `getDescriptor` on one of the fetched `STRUCT` objects. Only one `STRUCT` descriptor is needed for any one SQL object type.

Steps in Creating StructDescriptor and STRUCT Objects

To create a `STRUCT` object, you must:

1. Create a `StructDescriptor` object for the given Oracle object type, if it does not already exist.
2. Use the `StructDescriptor` to construct the `STRUCT` object.

A `StructDescriptor` is an instance of the `oracle.sql.StructDescriptor` class and describes a type of Oracle object. Only one `StructDescriptor` is necessary for each Oracle object type. The driver caches `StructDescriptor` objects to avoid re-creating them if the type has already been encountered.

Before you can construct a `STRUCT` object, a `StructDescriptor` must first exist for the given Oracle object type. If a `StructDescriptor` object does not exist, then you can create one by calling the static `StructDescriptor.createDescriptor` method. This method requires you to pass in the SQL type name of the Oracle object type and a connection object, as follows:

```
StructDescriptor structdesc = StructDescriptor.createDescriptor
                                        (sql_type_name, connection);
```

The *`sql_type_name`* parameter is a Java `String` containing the name of the Oracle object type, such as `EMPLOYEE`, and *`connection`* is the connection object.

Once you have your `StructDescriptor` object for the Oracle object type, you can construct the `STRUCT` object. To do this, provide the `Connection` object, the

StructDescriptor object, and an array of Java objects containing the attributes you want the STRUCT to contain.

The following constructors of STRUCT are available:

```
STRUCT(Connection conn, java.sql.StructDescriptor structDesc, Object[] attributes)

STRUCT(Connection conn, java.sql.StructDescriptor structDesc, java.util.Map map)
```

The structDesc parameter is the StructDescriptor object created previously and conn is your Connection object. The attributes can be passed as an array of java.lang.Object or as a java.util.Map object.

The following code illustrates the use of the constructor that takes an Object array:

```
...
Object[] attributes = {"attribute1", null};
STRUCT struct = new STRUCT(connection, structDescriptor, attributes);
...
```

The following code illustrates the use of the constructor that takes a Map object:

```
...
HashMap map = new HashMap(1);
map.put("A1","attribute1");
STRUCT struct = new STRUCT(connection, structDescriptor, map);
...
```

Using StructDescriptor Methods

A StructDescriptor can be thought of as a type object. This means that it contains information about the object type, including the type code, the type name, and how to convert to and from the given type. Remember, there should be only one StructDescriptor object for any one Oracle object type. You can then use that descriptor to create as many STRUCT objects as you need for that type.

The StructDescriptor class includes the following methods:

- getName

  This method returns the fully qualified SQL type name of the Oracle object.

- getLength

  This method returns the number of fields in the object type.

- getMetaData

  This method returns the meta data regarding this type. The returned ResultSetMetaData object contains the attribute name, attribute type code, and attribute type precision information. The column index in the ResultSetMetaData object maps to the position of the attribute in the STRUCT, with the first attribute being at index 1.

  See Also: "Functionality for Getting Object Meta Data" on page 15-35

Serializable STRUCT Descriptors

When you create a STRUCT object, you must first create a StructDescriptor object. You can do this by calling the StructDescriptor.createDescriptor method. The oracle.sql.StructDescriptor class is serializable. This means that you can write the complete state of a StructDescriptor object to an output stream for later

use. You can re-create the `StructDescriptor` object by reading its serialized state from an input stream. This is referred to as **deserializing**. With the `StructDescriptor` object serialized, you do not need to call the `StructDescriptor.createDescriptor` method, instead you deserialize the `StructDescriptor` object.

It is advisable to serialize a `StructDescriptor` object when the object type is complex but not changed often.

If you create a `StructDescriptor` object through deserialization, you must supply the appropriate database connection instance for the `StructDescriptor` object, using the `setConnection` method.

The following code provides the connection instance for a `StructDescriptor` object:

```
public void setConnection (Connection conn) throws SQLException
```

> **Note:** The JDBC driver does not verify that the connection object from the `setConnection` method connects to the same database from which the type descriptor was initially derived.

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 = new STRUCT (...);
ps.setObject(1, mySTRUCT, Types.STRUCT);
```

or:

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
STRUCT mySTRUCT = new STRUCT (...);
((OraclePreparedStatement)ps).setOracleObject(1, mySTRUCT);
```

STRUCT Automatic Attribute Buffering

The Oracle JDBC driver furnishes public methods to enable and disable buffering of `STRUCT` attributes.

> **See Also:** "ARRAY Automatic Element Buffering" on page 18-6

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 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.

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 will 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 `ORAData` and `ORADataFactory` interfaces provided by Oracle

The custom object class you create must implement one of these interfaces. The `ORAData` 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 `ORAData` 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 `ORAData` interface, and is able to generate classes that implement either one.

See Also: "Using JPublisher to Create Custom Object Classes" on page 15-32 and "Object-Type Inheritance" on page 15-21

This section covers the following topics:

- Relative Advantages of ORAData versus SQLData
- Understanding Type Maps for SQLData Implementations
- Creating Type Map and Defining Mappings for a SQLData Implementation
- Understanding the SQLData Interface
- Reading and Writing Data with a SQLData Implementation
- Understanding the ORAData Interface
- Reading and Writing Data with a ORAData Implementation

- Additional Uses for ORAData
- The Deprecated CustomDatum Interface

Relative Advantages of ORAData versus SQLData

In deciding which of the two interface implementations to use, you need to consider the advantages of `ORAData` and `SQLData`.

The `SQLData` interface is for mapping SQL objects only. The `ORAData` 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 a `ORAData` object from any data type found in Oracle Database. This could be useful, for example, for serializing `RAW` data in Java.

Advantages of ORAData

The advantages of `ORAData` 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 `ORAData` from an `oracle.sql.STRUCT`. This is more efficient because it avoids unnecessary conversions to native Java types.
- You can obtain the corresponding `Datum` object from the `ORAData` object, using the `toDatum` method.
- It provides better performance. `ORAData` works directly with `Datum` types, which is the internal format used by the driver to hold Oracle objects.

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 15-8

When using a SQLData implementation, if you do not include a type map entry, then the object will map to the `oracle.sql.STRUCT` class by default. `ORAData` implementations, by contrast, have their own mapping functionality so that a type map entry is not required. When using an `ORAData` implementation, use the Oracle `getORAData` method instead of the standard `getObject` 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.

Note: If you are migrating from Java Development Kit (JDK) 1.1.x to JDK 1.2.x, then you must ensure that your code uses a class that implements the `Map` interface. If you were using the `java.util.Hashtable` class under 1.1.x, then no change is necessary.

`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"));
   ```

   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 the 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, under JDK 1.2.x, also 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, 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, `setTypeMap` 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.sql.STRUCT` class. 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.sql.STRUCT` 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.

Understanding the SQLData Interface

One of the choices in making an Oracle object and its attribute data available to Java applications is to create a custom object class that implements the `SQLData` interface. Note that if you use this interface, you must supply a type map that specifies the Oracle object types in the database and the names of the corresponding custom object classes that you will create for them.

The `SQLData` interface defines methods that translate between SQL and Java for Oracle database objects. Standard JDBC provides a `SQLData` interface and companion `SQLInput` and `SQLOutput` interfaces in the `java.sql` package.

If you create a custom object class that implements `SQLData`, then you must provide a `readSQL` method and a `writeSQL` method, as specified by the `SQLData` interface.

The JDBC driver calls your `readSQL` method to read a stream of data values from the database and populate an instance of your custom object class. Typically, the driver would use this method as part of an `OracleResultSet` object `getObject` call.

Similarly, the JDBC driver calls your `writeSQL` method to write a sequence of data values from an instance of your custom object class to a stream that can be written to the database. Typically, the driver would use this method as part of an `OraclePreparedStatement` object `setObject` call.

Understanding the SQLInput and SQLOutput Interfaces

The JDBC driver includes classes that implement the `SQLInput` and `SQLOutput` interfaces. It is not necessary to implement the `SQLOutput` or `SQLInput` objects. The JDBC drivers will do this for you.

The `SQLInput` implementation is an input stream class, an instance of which is passed to the `readSQL` method. `SQLInput` includes a `read`*`XXX`* method for every possible Java type that attributes of an Oracle object may be converted to, such as `readObject`, `readInt`, `readLong`, `readFloat`, `readBlob`, and so on. Each `read`*`XXX`* method converts SQL data to Java data and returns it as the result with the corresponding Java type. For example, `readInt` returns an `int`.

The `SQLOutput` implementation is an output stream class, an instance of which is passed in to the `writeSQL` method. `SQLOutput` includes a `write`*`XXX`* method for each of these Java types. Each `write`*`XXX`* method converts Java data to SQL data, taking as input a parameter of the relevant Java type. For example, `writeString` would take as input a `String` attribute from your Java class.

Implementing readSQL and writeSQL Methods

When you create a custom object class that implements `SQLData`, you must implement the `readSQL` and `writeSQL` methods, as described here.

You must implement `readSQL` as follows:

```
public void readSQL(SQLInput stream, String sql_type_name) throws SQLException
```

- The `readSQL` method takes as input a `SQLInput` stream and a string that indicates the SQL type name of the data, that is, the name of the Oracle object type, such as `EMPLOYEE`.

When your Java application calls `getObject`, the JDBC driver creates a `SQLInput` stream object and populates it with data from the database. The driver can also determine the SQL type name of the data when it reads it from the database. When the driver calls `readSQL`, it passes in these parameters.

- For each Java data type that maps to an attribute of the Oracle object, `readSQL` must call the appropriate `read`*`XXX`* method of the `SQLInput` stream that is passed in.

  For example, if you are reading `EMPLOYEE` objects that have an employee name as a `CHAR` variable and an employee number as a `NUMBER` variable, then you must have a `readString` call and a `readInt` call in your `readSQL` method. JDBC calls these methods according to the order in which the attributes appear in the SQL definition of the Oracle object type.

- The `readSQL` method takes the data that the `read`*`XXX`* methods read and convert and assigns them to the appropriate fields or elements of a custom object class instance.

You must implement `writeSQL` as follows:

```
public void writeSQL(SQLOutput stream) throws SQLException
```

- The `writeSQL` method takes as input a `SQLOutput` stream.

  When your Java application calls `setObject`, the JDBC driver creates a `SQLOutput` stream object. When the driver calls `writeSQL`, it passes in this stream parameter.

- For each Java data type that maps to an attribute of the Oracle object, `writeSQL` must call the appropriate `write`*`XXX`* method of the `SQLOutput` stream that is passed in.

  For example, if you are writing to `EMPLOYEE` objects that have an employee name as a `CHAR` variable and an employee number as a `NUMBER` variable, then you must have a `writeString` call and a `writeInt` call in your `writeSQL` method. These methods must be called according to the order in which attributes appear in the SQL definition of the Oracle object type.

- The `writeSQL` method then writes the data to the `SQLOutput` stream by calling the `write`*`XXX`* methods so that it can be sent to the database once you execute the prepared statement.

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 your 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 getObject would return an oracle.sql.STRUCT object. Cast the output to type STRUCT, because the getObject method signature returns the generic java.lang.Object type.

```
if (rs.next())
   STRUCT empstruct = (STRUCT)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 an OracleCallableStatement instance, ocs, 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 an OracleCallableStatement to call the GETEMPLOYEE function, as follows:

```
OracleCallableStatement ocs = (OracleCallableStatement)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:

```
ocs.setInt(2, empnumber);
ocs.registerOutParameter(1, OracleTypes.STRUCT, "EMP_OBJECT");
ocs.execute();
```

3. Use the `getObject` method to retrieve the employee object. The following code assumes that there is a type map entry to map the Oracle object to the Java type `Employee`:

```
Employee emp = (Employee)ocs.getObject(1);
```

   If there is no type map entry, then `getObject` would return an `oracle.sql.STRUCT` object. Cast the output to the `STRUCT` type, because the `getObject` method returns an instance of the generic `java.lang.Object` class. This is done as follows:

```
STRUCT emp = (STRUCT)ocs.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 `OracleCallableStatement` to call the `addEmployee(?)` function.

```
OracleCallableStatement ocs =
  (OracleCallableStatement) conn.prepareCall("{ call addEmployee(?) }");
```

2. Use `setObject` to pass the `emp` object as an `IN` parameter to the callable statement. Then, call the statement.

```
ocs.setObject(1, emp);
ocs.execute();
```

Writing Data to an Oracle Object Using a SQLData Implementation

This 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);
```

   This statement uses the `emp` object and the `empname` and `empnumber` variables assigned in the preceding example.

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 (?)");
```

   This assumes `conn` is your connection object.

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 ORAData Interface

One of the choices in making an Oracle object and its attribute data available to Java applications is to create a custom object class that implements the `oracle.sql.ORAData` and `oracle.sql.ORADataFactory` interfaces. The `ORAData` and `ORADataFactory` interfaces are supplied by Oracle and are not a part of the JDBC standard.

> **Note:** The JPublisher utility supports the generation of classes that implement the `ORAData` and `ORADataFactory` interfaces.

Understanding ORAData Features

The `ORAData` interface has the following advantages:

- It recognizes Oracle extensions to the JDBC. `ORAData` uses `oracle.sql.Datum` types directly.
- It does not require a type map to specify the names of the Java custom classes you want to create.
- It provides better performance. `ORAData` works directly with `Datum` types, the internal format the driver uses to hold Oracle objects.

The `ORAData` and `ORADataFactory` interfaces do the following:

- The `toDatum` method of the `ORAData` class transforms the data into an `oracle.sql.*` representation.
- `ORADataFactory` specifies a `create` method equivalent to a constructor for your custom object class. It creates and returns an `ORAData` 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 an `oracle.sql.Datum` object and an integer indicating the corresponding SQL type code as specified in the `OracleTypes` class.

`ORAData` and `ORADataFactory` have the following definitions:

```
public interface ORAData
{
    Datum toDatum (OracleConnection conn) throws SQLException;
}

public interface ORADataFactory
{
    ORAData create (Datum d, int sql_Type_Code) throws SQLException;
}
```

Where *conn* represents the Connection object, *d* represents an object of type `oracle.sql.Datum` and *sql_Type_Code* represents the SQL type code of the `Datum` object.

Retrieving and Inserting Object Data

The JDBC drivers provide the following methods to retrieve and insert object data as instances of `ORAData`.

You can retrieve the object data in one of the following ways:

- Use the following `getORAData` method of the Oracle-specific `OracleResultSet` class:

  ```
  ors.getORAData (int col_index, ORADataFactory factory);
  ```

  This method takes as input the column index of the data in your result set and a `ORADataFactory` instance. For example, you can implement a `getORAFactory` method in your custom object class to produce the `ORADataFactory` instance to input to `getORAData`. The type map is not required when using Java classes that implement `ORAData`.

- Use the standard `getObject(index, map)` method specified by the `ResultSet` interface to retrieve data as instances of `ORAData`. 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 `setORAData` method of the Oracle-specific `OraclePreparedStatement` class:

  ```
  ops.setORAData (int bind_index, ORAData custom_obj);
  ```

  This method takes as input the parameter index of the bind variable and 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 `ORAData` instances without requiring a type map.

The following sections describe the `getORAData` and `setORAData` methods.

To continue the example of an Oracle object `EMPLOYEE`, you might have something like the following in your Java application:

```
ORAData datum = ors.getORAData(1, Employee.getORAFactory());
```

In this example, `ors` is an Oracle result set, `getORAData` is a method in the `OracleResultSet` class used to retrieve a `ORAData` object, and the `EMPLOYEE` is in column 1 of the result set. The `static Employee.getORAFactory` method will return a `ORADataFactory` 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.

Notes:

- `ORAData` and `ORADataFactory` are defined as separate interfaces so that different Java classes can implement them if you wish.
- To use the `ORAData` interface, your custom object classes must import `oracle.sql.*`.

Reading and Writing Data with a ORAData 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 `ORAData`.

Reading Data from an Oracle Object Using a ORAData Implementation

The following text summarizes the steps in reading data from an Oracle object into your Java application. These steps apply whether you implement `ORAData` 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 `getORAData` method of your Oracle result set to populate an instance of your custom object class with data from one row of the result set. The `getORAData` method returns an `oracle.sql.ORAData` object, which you can cast to your specific custom object class.

   ```
   if (ors.next())
      Employee emp = (Employee)ors.getORAData(1, Employee.getORAFactory());
   ```

   or:

   ```
   if (ors.next())
      ORAData datum = ors.getORAData(1, Employee.getORAFactory());
   ```

   This example assumes that `Employee` is the name of your custom object class and `ors` is the name of your `OracleResultSet` object.

   In case you do not want to use `getORAData`, the JDBC drivers let you use the `getObject` method of a standard JDBC `ResultSet` to retrieve `ORAData` data. However, 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.

   For example, if the SQL type name for your object is `EMPLOYEE`, then the corresponding Java class is `Employee`, which will implement `ORAData`. The corresponding Factory class is `EmployeeFactory`, which will implement `ORADataFactory`.

   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();
```

Note: Alternatively, you can fetch data using a callable statement object. The OracleCallableStatement class also has a getORAData method.

Writing Data to an Oracle Object Using a ORAData Implementation

The following text summarizes the steps in writing data to an Oracle object from your Java application. These steps apply whether you implement ORAData 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 setORAData method of the Oracle prepared statement to bind your Java data type object to the prepared statement.

   ```
   opstmt.setORAData(1, emp);
   ```

   The setORAData method calls the toDatum method of the custom object class instance to retrieve an oracle.sql.STRUCT object that can be written to the database.

   In this step you could also use the setObject method to bind the Java data type. For example:

   ```
   opstmt.setObject(1,emp);
   ```

   Note: You can use your Java data type objects as either IN or OUT bind variables.

Additional Uses for ORAData

The `ORAData` 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 `ORAData` 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.

It may be useful to provide custom Java classes to wrap `oracle.sql.*` types and perhaps 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 `ORAData` 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 `ORADataFactory` would have to implement a conversion from an object of type `oracle.sql.RAW` to the desired Java object. The `toDatum` method in `ORAData` would have to implement a conversion from the Java object to an `oracle.sql.RAW` object. This can be done, for example, by 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 `ORADataFactory` 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 `ORAData.toDatum` 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 `ORAData` 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 `ORAData`.

See Also: "Understanding the ORAData Interface" on page 15-16

The Deprecated CustomDatum Interface

After the `oracle.jdbc` interfaces were introduced in Oracle9*i* Database as an alternative to the `oracle.jdbc.driver` classes, the `oracle.sql.CustomDatum` and `oracle.sql.CustomDatumFactory` interfaces, formerly used to access

customized objects, were deprecated. Oracle recommends you use the new interfaces, `oracle.sql.ORAData` and `oracle.sql.ORADataFactory`.

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, the Oracle JDBC driver returns a Java object that represents the `STUDENT_T` object.

This section covers the following topics:

- Creating Subtypes
- Implementing Customized Classes for Subtypes
- Retrieving Subtype Objects
- Creating Subtype Objects
- Sending Subtype Objects
- Accessing Subtype Data Fields
- Inheritance Meta Data 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 for:

```
PERSON_T
|
STUDENT_T
|
PARTTIMESTUDENT_T
```

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 Application Developer's Guide - Object-Relational Features*

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 `ORAData` or `SQLData` solution in creating classes to map to the hierarchy of object types.

This section covers the following topics:

- Use of ORAData for Type Inheritance Hierarchy
- Use of SQLData for Type Inheritance Hierarchy
- JPublisher Utility

Use of ORAData for Type Inheritance Hierarchy

Customized mapping where Java classes implement the `oracle.sql.ORAData` interface is the recommended mapping. `ORAData` mapping requires the JDBC application to implement the `ORAData` and `ORADataFactory` interfaces. The class implementing the `ORADataFactory` interface contains a factory method that produces objects. Each object represents a database object.

The hierarchy of the class implementing the `ORAData` 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 ORAData

Code for the `Person.java` class which implements the `ORAData` and `ORADataFactory` interfaces:

```
class Person implements ORAData, ORADataFactory
{
  static final Person _personFactory = new Person();

  public NUMBER ssn;
  public CHAR name;
  public CHAR address;

  public static ORADataFactory getORADataFactory()
```

```
  {
    return _personFactory;
  }

  public Person () {}

  public Person(NUMBER ssn, CHAR name, CHAR address)
  {
    this.ssn = ssn;
    this.name = name;
    this.address = address;
  }

  public Datum toDatum(OracleConnection c) throws SQLException
  {
    StructDescriptor sd =
      StructDescriptor.createDescriptor("SCOTT.PERSON_T", c);
    Object [] attributes = { ssn, name, address };
    return new STRUCT(sd, c, attributes);
  }

  public ORAData create(Datum d, 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 ORADataFactory getORADataFactory()
  {
    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 Datum toDatum(OracleConnection c) throws SQLException
  {
    StructDescriptor sd =
```

```
      StructDescriptor.createDescriptor("SCOTT.STUDENT_T", c);
    Object [] attributes = { ssn, name, address, deptid, major };
    return new STRUCT(sd, c, attributes);
  }

  public CustomDatum create(Datum d, 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 `ORAData` 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`.

ORADataFactory 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())
{
  Object s = rset.getORAData (1, PersonFactory.getORADataFactory());
  ...
}
```

A class implementing the `ORADataFactory` 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.getORADataFactory` 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 ORADataFactory
{
  static final PersonFactory _factory = new PersonFactory ();

  public static ORADataFactory getORADataFactory()
  {
    return _factory;
  }

  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    STRUCT s = (STRUCT) d;
    if (s.getSQLTypeName ().equals ("SCOTT.PERSON_T"))
      return Person.getORADataFactory ().create (d, sqlType);
    else if (s.getSQLTypeName ().equals ("SCOTT.STUDENT_T"))
      return Student.getORADataFactory ().create(d, sqlType);
    else if (s.getSQLTypeName ().equals ("SCOTT.PARTTIMESTUDENT_T"))
      return ParttimeStudent.getORADataFactory ().create(d, 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, 'Scott', '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`, `ORAData`, and `ORADataFactory` 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`, `ORAData`, and `ORADataFactory` interfaces, can mirror the inheritance hierarchy.

See Also:

- "Using JPublisher to Create Custom Object Classes" on page 15-32
- *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 `ORAData` mapping to retrieve a subtype.

Using Default Mapping

By default, a database object is returned as an instance of the `oracle.sql.STRUCT` class. This instance may represent an object of either the declared type or subtype of the declared type. If the `STRUCT` class represents a subtype object in the database, then it contains the attributes of its supertype as well as those defined in the subtype.

The Oracle JDBC driver returns database objects in their most specific type. The JDBC application can use the `getSQLTypeName` method of the `STRUCT` class 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 SCOTT.PERSON_T, SCOTT.STUDENT_T or SCOTT.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 wrapper 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 ("SCOTT.PERSON_T", Class.forName ("Person"));
map.put ("SCOTT.STUDENT_T", Class.forName ("Student"));
map.put ("SCOTT.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 ORAData Mapping

With `ORAData` mapping, the JDBC driver returns the database object as an instance of the class implementing the `ORAData` interface.

The 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 the Oracle JDBC drivers:

- The JDBC application uses the `getORAData(int idx, ORADataFactory f)` method to access database objects. The second parameter of the `getORAData` method specifies an instance of the factory class that produces the customized class. The `getORAData` method is available in the `OracleResultSet` and `OracleCallableStatement` classes.
- 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.getORAData (1, PersonFactory.getORADataFactory());
  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`- or `ORAData`-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
StructDescriptor desc = StructDescriptor.createDescriptor
("SCOTT.PARTTIMESTUDENT", conn);
Object[] attrs = {
  new Integer(1234), "Scott", "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 = new STRUCT (desc, conn, 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 ORAData interface. The 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 normal 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.sql.STRUCT 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, as listed in Table 5–1. 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, as listed in Table 5–1. 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 ("SCOTT.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 ("SCOTT.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 ("SCOTT.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 Meta Data Methods

Oracle JDBC drivers provide a set of meta data methods to access inheritance properties. The inheritance meta data methods are defined in the `oracle.sql.StructDescriptor` and `oracle.jdbc.StructMetaData` classes.

The `oracle.sql.StructDescriptor` class provides the following inheritance meta data methods:

- `String[] getSubtypeNames()`

  Returns the SQL type names of the direct subtypes.

- `boolean isFinalType()`

Indicates whether the object type is a final type. An object type is `FINAL` if no subtypes can be created for this type; the default is `FINAL`, and a type declaration must have the `NOT FINAL` keyword to be subtypable.

- `boolean isSubTyp()`

  Indicates whether the object type is a subtype.

- `boolean isInstantiable()`

  Indicates whether the object type is instantiable. An object type is `NOT INSTANTIABLE` if it is not possible to construct instances of this type.

- `String getSupertypeName()`

  Returns the SQL type names of the direct supertype.

- `int getLocalAttributeCount()`

  Returns the number of attributes defined in the subtype.

The `StructMetaData` class provides inheritance meta data 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 meta data methods:

- `int getLocalColumnCount()`

  Returns the number of attributes defined in the subtype, which is similar to the `getLocalAttributeCount()` method of the `StructDescriptor` class.

- `boolean isInherited(int column)`

  Indicates whether the attribute is inherited. The value for the `column` attribute begins from `1`.

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 `ORAData` interface, according to how you set the JPublisher type mappings.

If you use the `ORAData` 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 `ORAData` 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 a `ORAData` 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 a Oracle object type:

- A setting of `-usertypes=oracle`, which is the default setting, instructs JPublisher to create a `ORAData` implementation for the custom object class. This will also result in JPublisher producing a `ORAData` 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` or `oracle.sql.REF` for the reference type.

  Note: You can also use JPublisher with a `-usertypes=oracle` setting in creating `ORAData` 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 15–1 summarizes JPublisher categories for SQL types, the mapping settings relevant for each category, and the default settings.

Table 15–1 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 Meta Data
- Steps for Retrieving Object Meta Data

Functionality for Getting Object Meta Data

The `oracle.sql.StructDescriptor` class includes functionality to retrieve meta data 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 meta data 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 meta data in the same way that the standard `java.sql.ResultSetMetaData` interface specifies support for result set meta data.

The `StructMetaData` class includes the following standard methods that are also specified by `ResultSetMetaData`:

- `String getColumnName(int column) throws SQLException`

  This returns a `String` that specifies the name of the specified attribute, such as `salary`.

- `int getColumnType(int column) throws SQLException`

  This returns an `int` that specifies the type code of the specified attribute, according to the `java.sql.Types` and `oracle.jdbc.OracleTypes` classes.

- `String getColumnTypeName(int column) throws SQLException`

  This returns a string that specifies the type of the specified attribute, such as `BigDecimal`.

- `int getColumnCount() throws SQLException`

  This returns the number of attributes in the object type.

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` class `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 Meta Data

Use the following steps to obtain meta data 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 meta data 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 meta data
  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 meta data
      getAttributeInfo (conn, attr_typeName);
    }
  }
}
```

16

Working with LOBs and BFILEs

This chapter describes how to use Java Database Connectivity (JDBC) and the `oracle.sql.*` classes to access and manipulate large object (LOB) and binary file (BFILE) locators and data. This chapter contains the following sections:

- Oracle Extensions for LOBs and BFILEs
- Working with BLOBs and CLOBs
- Data Interface for LOBs
- Working With Temporary LOBs
- Using Open and Close With LOBs
- Working with BFILEs

Notes:

- In Oracle Database 10*g*, the Oracle JDBC drivers support the JDBC 3.0 `java.sql.Clob` and `java.sql.Blob` interfaces. Certain Oracle extensions made in `oracle.sql.CLOB` and `oracle.sql.BLOB` in earlier Oracle Database releases are no longer necessary and are deprecated. You should port your application to the standard JDBC 3.0 interface.
- Prior to Oracle Database 10*g*, the maximum size of a LOB was 2^{32} bytes. This restriction is removed in Oracle Database 10*g*, and the maximum size is now limited to the size of available physical storage. The Java LOB application programming interface (API) has not changed.

Oracle Extensions for LOBs and BFILEs

LOBs are stored in a way that optimizes space and provides efficient access. The JDBC drivers provide support for two types of LOB: binary large object (BLOB), which is used for unstructured binary data, and character large object (CLOB), which is used for character data. BLOB and CLOB data is accessed and referenced by using a locator that is stored in the database table and points to the BLOB or CLOB data, which is outside the table.

BFILEs are large binary data objects stored in operating system files outside of database tablespaces. These files use reference semantics. They can also be located on tertiary storage devices, such as hard disks, CD-ROMs, PhotoCDs, and DVDs. As with BLOBs and CLOBs, a BFILE is accessed and referenced by a locator which is stored in the database table and points to the BFILE data.

To work with LOB data, you must first obtain a LOB locator. Then you can read or write LOB data and perform data manipulation.

The JDBC drivers support the following `oracle.sql.*` classes for BLOBs, CLOBs, and BFILEs:

- `oracle.sql.BLOB`
- `oracle.sql.CLOB`
- `oracle.sql.BFILE`

The `oracle.sql.BLOB` and `oracle.sql.CLOB` classes implement the `java.sql.Blob` and `java.sql.Clob` interfaces, respectively. In contrast, `BFILE` is an Oracle extension, without a corresponding `java.sql` interface.

Instances of these classes contain only the locators for these data types, not the data. After accessing the locators, you must perform some additional steps to access the data.

Note: Your code cannot call the `BLOB`, `CLOB`, or `BFILE` constructors. To create a new LOB use the `createTemporary`, `getEmptyCLOB`, or `getEmptyBLOB` methods.

Working with BLOBs and CLOBs

This section describes how to read and write data to and from BLOBs and CLOBs in Oracle Database, using LOB locators. This section covers the following topics:

- Getting and Passing BLOB and CLOB Locators
- Reading and Writing BLOB and CLOB Data
- Creating and Populating a BLOB or CLOB Column
- Accessing and Manipulating BLOB and CLOB Data
- Additional BLOB and CLOB Features

Getting and Passing BLOB and CLOB Locators

Standard as well as Oracle-specific getter and setter methods are available for retrieving or passing LOB locators from or to the database. This section covers the following topics:

- Retrieving BLOB and CLOB Locators
- Passing BLOB and CLOB Locators

Retrieving BLOB and CLOB Locators

Given a standard JDBC result set or callable statement that includes BLOB or CLOB locators, you can access the locators by using standard getter methods. You can use the standard `getBlob` and `getClob` methods, which return `java.sql.Blob` and `Clob` objects, respectively.

Note: All the standard and Oracle-specific getter methods discussed here take either an `int` column index or a `String` column name as input.

If you retrieve or cast the result set or the callable statement to `OracleResultSet` or `OracleCallableStatement`, then you can use Oracle extensions, as follows:

- You can use `getBLOB` and `getCLOB`, which return `oracle.sql.BLOB` and `CLOB` objects, respectively.
- You can also use the `getOracleObject` method, which returns an `oracle.sql.Datum` object, and cast the output appropriately.

Example: Getting BLOB and CLOB Locators from a Result Set

Assume the database has a table called `lob_table` with a column for a BLOB locator, `blob_col`, and a column for a CLOB locator, `clob_col`. This example assumes that you have already created the `Statement` object, `stmt`.

First, select the LOB locators into a standard result set, then get the LOB data into appropriate Java classes:

```
// Select LOB locator into standard result set.
ResultSet rs =
   stmt.executeQuery ("SELECT blob_col, clob_col FROM lob_table");
while (rs.next())
{
   // Get LOB locators into Java wrapper classes.
   java.sql.Blob blob = (java.sql.Blob)rs.getObject(1);
   java.sql.Clob clob = (java.sql.Clob)rs.getObject(2);
   (...process...)
}
```

The output is cast to `java.sql.Blob` and `java.sql.Clob`. As an alternative, you can cast the output to `oracle.sql.BLOB` and `oracle.sql.CLOB` to take advantage of extended functionality offered by the `oracle.sql.*` classes. For example, you can rewrite the preceding code to get the LOB locators as:

```
   // Get LOB locators into Java wrapper classes.
   oracle.sql.BLOB blob = (BLOB)rs.getObject(1);
   oracle.sql.CLOB clob = (CLOB)rs.getObject(2);
   (...process...)
```

Example: Getting a CLOB Locator from a Callable Statement

The callable statement methods for retrieving LOBs are identical to the result set methods.

For example, if you have an `OracleCallableStatement` instance, `ocs`, that calls a function `func` that has a CLOB output parameter, then set up the callable statement as in the following example.

This example registers `OracleTypes.CLOB` as the type code of the output parameter.

```
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("{? = call func()}");
ocs.registerOutParameter(1, OracleTypes.CLOB);
ocs.execute();
oracle.sql.CLOB clob = ocs.getCLOB(1);
```

Passing BLOB and CLOB Locators

Given a standard JDBC prepared statement or callable statement, you can use standard setter methods to pass LOB locators. These methods are defined as follows:

```
public void setBlob(int index, Blob value);
public void setClob(int index, Clob value);
```

> **Note:** If you pass a BLOB to a PL/SQL procedure, then the BLOB must be no bigger than 32K - 7. If you pass a BLOB that exceeds this limit, then you will receive a `SQLException`.

Given an Oracle-specific `OraclePreparedStatement` or `OracleCallableStatement`, then you can use Oracle extensions as follows:

- Use `setBLOB` and `setCLOB`, which take `oracle.sql.BLOB` and `CLOB` locators as input, respectively.
- Use the `setOracleObject` method, which simply specifies an `oracle.sql.Datum` input.

Example: Passing a BLOB Locator to a Prepared Statement

If you have an `OraclePreparedStatement` object `ops` and a BLOB named `my_blob`, then write the BLOB to the database as follows:

```
OraclePreparedStatement ops = (OraclePreparedStatement)conn.prepareStatement
                           ("INSERT INTO blob_table VALUES(?)");
ops.setBLOB(1, my_blob);
ops.execute();
```

Example: Passing a CLOB Locator to a Callable Statement

If you have an `OracleCallableStatement` object `ocs` and a CLOB named `my_clob`, then input the CLOB to the stored procedure `proc` as follows:

```
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("{call proc(?)}");
ocs.setClob(1, my_clob);
ocs.execute();
```

Reading and Writing BLOB and CLOB Data

Once you have a LOB locator, you can use JDBC methods to read and write the LOB data. LOB data is materialized as a Java array or stream. However, unlike most Java streams, a locator representing the LOB data is stored in the table. Thus, you can access the LOB data at any time during the life of the connection.

To read and write the LOB data, use the methods in the `oracle.sql.BLOB` or `oracle.sql.CLOB` class, as appropriate. These classes provide functionality such as reading from the LOB into an input stream, writing from an output stream into a LOB, determining the length of a LOB, and closing a LOB.

> **Notes:** To write LOB data, the application must acquire a write lock on the LOB object. One way to accomplish this is through a `SELECT FOR UPDATE`. Also, you must disable auto-commit mode.

To read and write LOB data, you can use these methods:

- To read from a BLOB, use the `getBinaryStream` method of an `oracle.sql.BLOB` object to retrieve the entire `BLOB` as an input stream. This returns a `java.io.InputStream` object.

  As with any `InputStream` object, use one of the overloaded `read` methods to read the LOB data and use the `close` method when you finish.

- To write to a BLOB, use the `setBinaryStream` method of an `oracle.sql.BLOB` object to retrieve the BLOB as an output stream. This returns a `java.io.OutputStream` object to be written back to the BLOB.

  As with any `OutputStream` object, use one of the overloaded `write` methods to update the LOB data and use the `close` method when you finish.

- To read from a CLOB, use the `getAsciiStream` or `getCharacterStream` method of an `oracle.sql.CLOB` object to retrieve the entire CLOB as an input stream. The `getAsciiStream` method returns an ASCII input stream in a `java.io.InputStream` object. The `getCharacterStream` method returns a `Unicode` input stream in a `java.io.Reader` object.

  As with any `InputStream` or `Reader` object, use one of the overloaded `read` methods to read the LOB data and use the `close` method when you finish.

  You can also use the `getSubString` method of `oracle.sql.CLOB` object to retrieve a subset of the CLOB as a character string of type `java.lang.String`.

- To write to a CLOB, use the `setAsciiStream` or `setCharacterStream` method of an `oracle.sql.CLOB` object to retrieve the CLOB as an output stream to be written back to the CLOB. The `setAsciiStream` method returns an ASCII output stream in a `java.io.OutputStream` object. The `setCharacterStream` method returns a Unicode output stream in a `java.io.Writer` object.

  As with any `Stream` or `Writer` object, use one of the overloaded `write` methods to update the LOB data and use the `flush` and `close` methods when you finish.

Notes:

- The stream write methods described in this section write directly to the database when you write to the output stream. You do *not* need to run an `UPDATE` to write the data. However, you need to call `close` or `flush` to ensure all changes are written. CLOBs and BLOBs are transaction controlled. After writing to either, you must commit the transaction for the changes to be permanent. BFILEs are not transaction controlled. Once you write to them the changes are permanent, even if the transaction is rolled back, unless the external file system does something else.
- When writing to or reading from a CLOB, the JDBC drivers perform all character set conversions for you.

Example: Reading BLOB Data

Use the `getBinaryStream` method of the `oracle.sql.BLOB` class to read BLOB data. The `getBinaryStream` method provides access to the BLOB data through a binary stream.

The following example uses the `getBinaryStream` method to read BLOB data through a byte stream and then reads the byte stream into a byte array, returning the number of bytes read, as well.

```
// Read BLOB data from BLOB locator.
InputStream byte_stream = my_blob.getBinaryStream(1L);
byte [] byte_array = new byte [10];
int bytes_read = byte_stream.read(byte_array);
...
```

Example: Reading CLOB Data

The following example uses the `getCharacterStream` method to read `CLOB` data into a Unicode character stream. It then reads the character stream into a character array, returning the number of characters read, as well.

```
// Read CLOB data from CLOB locator into Reader char stream.
Reader char_stream = my_clob.getCharacterStream(1L);
char [] char_array = new char [10];
int chars_read = char_stream.read (char_array, 0, 10);
...
```

The next example uses the `getAsciiStream` method of the `oracle.sql.CLOB` class to read CLOB data through an ASCII character stream. It then reads the ASCII stream into a byte array, returning the number of bytes read, as well.

```
// Read CLOB data from CLOB locator into Input ASCII character stream
Inputstream asciiChar_stream = my_clob.getAsciiStream(1L);
byte[] asciiChar_array = new byte[10];
int asciiChar_read = asciiChar_stream.read(asciiChar_array,0,10);
```

Example: Writing BLOB Data

Use the `setBinaryOutputStream` method of an `oracle.sql.BLOB` object to write BLOB data.

The following example reads a vector of data into a byte array, then uses the `setBinaryOutputStream` method to write an array of character data to a BLOB.

```
java.io.OutputStream outstream;

// read data into a byte array
byte[] data = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// write the array of binary data to a BLOB
outstream = ((BLOB)my_blob).setBinaryOutputStream(1L);
outstream.write(data);
...
```

Example: Writing CLOB Data

Use the `setCharacterStream` method or the `setAsciiStream` method to write data to a CLOB. The `setCharacterStream` method returns a `Unicode` output stream. The `setAsciiStream` method returns an ASCII output stream.

The following example reads a vector of data into a character array, then uses the `setCharacterStream` method to write the array of character data to a CLOB.

```
java.io.Writer writer;

// read data into a character array
char[] data = {'0','1','2','3','4','5','6','7','8','9'};

// write the array of character data to a CLOB
writer = ((CLOB)my_clob).setCharacterStream();
writer.write(data);
writer.flush();
```

```
writer.close();
...
```

The next example reads a vector of data into a byte array, then uses the `setAsciiStream` method to write the array of ASCII data to a CLOB.

```
java.io.OutputStream out;

// read data into a byte array
byte[] data = {'0','1','2','3','4','5','6','7','8','9'};

// write the array of ascii data to a CLOB
out = clob.setAsciiStream();
out.write(data);
out.flush();
out.close();
```

Creating and Populating a BLOB or CLOB Column

Create and populate a `BLOB` or `CLOB` column in a table by using SQL statements.

> **Note:** You cannot construct a new `BLOB` or `CLOB` locator in your application with a Java `new` statement. You must create the locator through a SQL operation, and then select it into your application or with the `createTemporary` or `empty_lob` methods.

Create a `BLOB` or `CLOB` column in a table with the SQL `CREATE TABLE` statement, then populate the LOB. This includes creating the LOB entry in the table, obtaining the LOB locator, and then copying the data into the LOB.

Creating a BLOB or CLOB Column in a New Table

To create a `BLOB` or `CLOB` column in a new table, run the SQL `CREATE TABLE` statement. The following example code creates a `BLOB` column in a new table. This example assumes that you have already created your `Connection` object `conn` and `Statement` object `stmt`:

```
String cmd = "CREATE TABLE my_blob_table (x VARCHAR2 (30), c BLOB)";
stmt.execute (cmd);
```

In this example, the `VARCHAR2` column designates a row number, such as 1 or 2, and the `BLOB` column stores the locator of the BLOB data.

Populating a BLOB or CLOB Column in a New Table

This example demonstrates how to populate a `BLOB` or `CLOB` column by reading data from a stream. These steps assume that you have already created your `Connection` object `conn` and `Statement` object `stmt`. The table `my_blob_table` is the table that was created in the previous section.

The following example writes the `john.gif` file to a BLOB:

1. Begin by using SQL statements to create the BLOB entry in the table. Use the `empty_blob` function to create the BLOB locator.

   ```
   stmt.execute ("INSERT INTO my_blob_table VALUES ('row1', empty_blob())");
   ```

2. Get the BLOB locator from the table.

   ```
   BLOB blob;
   ```

```
cmd = "SELECT * FROM my_blob_table WHERE X='row1' FOR UPDATE";
ResultSet rset = stmt.executeQuery(cmd);
rset.next();
BLOB blob = ((OracleResultSet)rset).getBLOB(2);
```

Note: You must disable auto-commit mode.

3. Declare a file handler for the `john.gif` file, then print the length of the file. This value will be used later to ensure that the entire file is read into the BLOB. Next, create a `FileInputStream` object to read the contents of the file, and an `OutputStream` object to retrieve the BLOB as a stream.

```
File binaryFile = new File("john.gif");
System.out.println("john.gif length = " + binaryFile.length());
FileInputStream instream = new FileInputStream(binaryFile);
OutputStream outstream = blob.setBinaryStream(1L);
```

4. Call `getBufferSize` to retrieve the ideal buffer size to use in writing to the BLOB, then create the `buffer` byte array.

```
int size = blob.getBufferSize();
byte[] buffer = new byte[size];
int length = -1;
```

5. Use the `read` method to read the file to the byte array `buffer`, then use the `write` method to write it to the BLOB. When you finish, close the input and output streams and commit the changes.

```
while ((length = instream.read(buffer)) != -1)
   outstream.write(buffer, 0, length);
instream.close();
outstream.close();
conn.commit();
```

Once your data is in the BLOB or CLOB, you can manipulate the data.

Accessing and Manipulating BLOB and CLOB Data

Once you have your BLOB or CLOB locator in a table, you can access and manipulate the data to which it points. To access and manipulate the data, you first must select their locators from a result set or from a callable statement.

After you select the locators, you can retrieve the BLOB or CLOB data. You will usually want to cast the result set to `OracleResultSet`, so that you can retrieve the data in `oracle.sql.*` format. After retrieving the BLOB or CLOB data, you can manipulate it however you want.

This example is a continuation of the example in the previous section. It uses the SQL `SELECT` statement to select the BLOB locator from the table `my_blob_table` into a result set. The result of the data manipulation is to print the length of the BLOB in bytes.

```
// Select the blob - what we are really doing here
// is getting the blob locator into a result set
BLOB blob;
cmd = "SELECT *  FROM my_blob_table";
ResultSet rset = stmt.executeQuery (cmd);
```

```
// Get the blob data - cast to OracleResult set to
// retrieve the data in oracle.sql format
String index = ((OracleResultSet)rset).getString(1);
blob = ((OracleResultSet)rset).getBLOB(2);

// get the length of the blob
int length = blob.length();

// print the length of the blob
System.out.println("blob length" + length);

// read the blob into a byte array
// then print the blob from the array
byte bytes[] = blob.getBytes(1, length);
blob.printBytes(bytes, length);
```

Additional BLOB and CLOB Features

In addition to what has already been discussed in this chapter, the `oracle.sql.BLOB` and `oracle.sql.CLOB` classes have a number of methods for further functionality.

Note: The `oracle.sql.CLOB` class supports all the character sets that the Oracle data server supports for CLOB types.

See Also: "Working With Temporary LOBs" on page 16-14 and "Using Open and Close With LOBs" on page 16-16

Additional BLOB Methods

The `oracle.sql.BLOB` class includes the following methods:

- `close`

  Closes the BLOB associated with the locator.

- `freeTemporary`

  Frees the storage used by a temporary BLOB.

- `getBufferSize`

  Returns the ideal buffer size, according to calculations by the JDBC driver, to use in reading and writing BLOB data. This value is a multiple of the chunk size and is close to 32K.

- `getBytes`

  Reads from the BLOB data, starting at a specified point, into a supplied buffer. The method returns a `byte` array which holds the contents of the BLOB.

Notes: In releases prior to Oracle Database 10*g* release 2 (10.2), the `getBytes` method when used on a `BLOB` column or output parameter incorrectly returns the bytes of the `LOB` locator, which is internal data and should not be used by applications. However, in Oracle Database 10*g* release 2 (10.2) the `getBytes` method returns a `byte` array that holds the contents of the `BLOB`.

This method can give errors for `BLOB` data that exceed available memory or the size imposed by the Java language.

- `getChunkSize`

  Returns the Oracle chunking size, which can be specified by the database administrator when the LOB column is first created. This value, in Oracle blocks, determines the size of the chunks of data read or written by the LOB data layer in accessing or modifying the BLOB value. Part of each chunk stores system-related information, and the rest stores LOB data. Performance is enhanced if read and write requests use some multiple of the chunk size.

- `isOpen`

  Returns `true` if the BLOB was opened by calling the `open` method; otherwise, it returns `false`.

- `isTemporary`

  Returns true if the BLOB is a temporary BLOB.

- `length`

  Returns the length of the BLOB in bytes.

- `open`

  Opens the BLOB associated with the locator.

- `open(int)`

  Opens the BLOB associated with the locator in the mode specified by the argument.

- `position(byte[], long)`

  Determines the byte position in the BLOB where a given pattern begins.

- `setBinaryStream(long)`

  Returns the BLOB data for this `Blob` instance as a stream of bytes beginning at the position in the BLOB specified in the argument.

- `setBytes(long, byte[])`

  Writes BLOB data, starting at a specified point, from a supplied buffer.

- `truncate(long)`

  Trims the value of the BLOB to the length specified by the argument.

Additional CLOB Methods

The `oracle.sql.CLOB` class includes the following methods:

- `close`

  Closes the CLOB associated with the locator.

- `freeTemporary`

  Frees the storage used by a temporary CLOB.

- `getAsciiStream`

  Returns the CLOB value designated by the `Clob` object as a stream of ASCII bytes.

- `getAsciiStream(long)`

  Returns the CLOB value designated by the CLOB object as a stream of ASCII bytes, beginning at the position in the CLOB specified by the argument.

- `getBufferSize`

  Returns the ideal buffer size, according to calculations by the JDBC driver, to use in reading and writing CLOB data. This value is a multiple of the chunk size and is close to 32K.

- `getCharacterStream`

  Returns the CLOB data as a stream of Unicode characters.

- `getCharacterStream(long)`

  Returns the CLOB data as a stream of Unicode characters beginning at the position in the CLOB specified by the argument.

- `getChars`

  Retrieves characters from a specified point in the CLOB data into a character array.

- `getChunkSize`

  Returns the Oracle chunking size, which can be specified by the database administrator when the LOB column is first created. This value, in Oracle blocks, determines the size of the chunks of data read or written by the LOB data layer in accessing or modifying the CLOB value. Part of each chunk stores system-related information and the rest stores LOB data. Performance is enhanced if you make read and write requests using some multiple of the chunk size.

- `getSubString(long, int)`

  Retrieves a substring from a specified point in the CLOB data.

- `isOpen`

  Returns `true` if the CLOB was opened by calling the `open` method; otherwise, it returns `false`.

- `isTemporary`

  Returns `true` if and only if the CLOB is a temporary CLOB.

- `length`

  Returns the length of the CLOB in characters.

- `open`

  Opens the CLOB associated with the locator.

- `open(int)`

  Opens the CLOB associated with the locator in the mode specified by the argument.

- `position(String, long)`

Determines the character position in the CLOB at which a given substring begins.

- `putChars(long, char[])`

  Writes characters from a character array to a specified point in the CLOB data.

- `setAsciiStream(long)`

  Returns a `java.io.OutputStream` object to write data to the CLOB as a stream. The data is written beginning at the position in the CLOB specified by the argument.

- `setCharacterStream(long)`

  Returns a `java.io.Writer` object to write data to the CLOB as a stream. The data is written beginning at the position in the CLOB specified by the argument.

- `setString(long pos, String str)`

  Writes a string to a specified point in the CLOB data.

- `truncate(long)`

  Trims the value of the CLOB to the length specified by the argument.

Creating Empty LOBs

Before writing data to an internal LOB, you must make sure the LOB column/attribute is not `null`. It must contain a locator. You can accomplish this by initializing the internal LOB as an empty LOB in an `INSERT` or `UPDATE` statement, using the `getEmpty`*`XXX`* methods defined in the `oracle.sql.BLOB` and `oracle.sql.CLOB` classes:

- `public static BLOB getEmptyBLOB() throws SQLException`
- `public static CLOB getEmptyCLOB() throws SQLException`

A JDBC driver creates an empty `LOB` instance without making database round trips. You can use empty LOBs in the following:

- `set`*`XXX`* methods of the `OraclePreparedStatement` class
- `update`*`XXX`* methods of updatable result sets
- attributes of `STRUCT` objects
- elements of `ARRAY` objects

> **Note:** Because a `getEmpty`*`XXX`* method creates a special marker that does not contain a locator, a JDBC application cannot read or write to it. The JDBC driver throws the exception `ORA-17098 Invalid empty LOB operation`, if a JDBC application attempts to read or write to an empty LOB before it is stored in the database.

Data Interface for LOBs

The data interface for LOBs provides a streamlined mechanism for writing and reading the entire LOB contents. It is simpler to code and faster in many cases. It does not provide the random access capability or access beyond 2147483648 elements as do the standard `java.sql.Blob` and `java.sql.Clob` interfaces and the Oracle extensions, `oracle.sql.BLOB`, `oracle.sql.BFILE`, and `oracle.sql.CLOB`.

Input

In Oracle Database 10*g* release 2 (10.2), the `setBytes`, `setBinaryStream`, `setString`, `setCharacterStream`, and `setAsciiStream` methods of `PreparedStatement` are extended for `BLOB` and `CLOB` parameters.

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 limit on the length specified for the stream functions except the limits imposed by the Java language, which is that array sizes are limited to positive Java `int` or 2147483648 elements.

For the server side internal driver there is currently a limitation of 4000 bytes for operations on SQL statements, such as an `INSERT` statement. The limitation does not apply for PL/SQL statements. There is a simple workaround for an `INSERT` statement, which is to wrap it in a PL/SQL block, as follows:

```
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:

- For SQL statements:
  - `setBytes` switches to `setBinaryStream` for data larger than 2000 bytes
  - `setString` switches to `setCharacterStream` for data larger than 32766 characters
- PL/SQL statements
  - `setBytes` switches to `setBinaryStream` for data larger than 2000 bytes and to `setBytesForBlob` for data larger that 32512 bytes
  - `setString` switches to `setStringForClob` for string data larger than 32512 bytes in the database character set or national character set depending on whether `setFormOfUse` has been used for `NCLOB` parameters

This will have impact on some programs, which formerly got `ORA-17157` errors for attempts to use `setString` for `String` values larger than 32766 characters. Now, depending on the type of the target parameter an error may occur while the application 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.

Oracle Database 10*g* release 1 (10.1) has the `SetBigStringTryClob` connection property. Setting this property causes the standard `setString` method to switch to `setStringForClob` method for large data. This property is no longer used or needed. The `setBytesForBlob` and `setStringForClob` methods create temporary LOBs, which are automatically freed when the statement is executed or closed before execution.

However, when a PL/SQL procedure or function is embedded in a SQL statement, data less than 4 KB is bound as `String`, which is the standard. When data is greater than 4KB, the driver binds the data as a `String` as for any SQL statement. This will throw an error. The workaround is to use `setClob` or `setCharacterStream` instead of `setString` or `setStringForClob`. You can also create a callable statement.

Output

The `getBytes`, `getBinaryStream`, `getSting`, `getCharacterStream`, and `getAsciiStream` methods of `ResultSet` and `CallableStatement` are extended to work with `BLOB`, `CLOB`, and `BFILE` columns or `OUT` parameters. These methods will work for any LOB of length less than 2147483648. This operates entirely on the client-side and will work with any supported version of the database, that is, Oracle Database 8.1.7 and later.

`BLOB`, `BFILE`, or `CLOB` data can be read and written using the same streaming mechanism as for `LONG RAW` and `LONG` data. To read, use `defineColumnType(nn, Types.LONGVARBINARY)` or `defineColumnType(nn,Types.LONGVARCHAR)` 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` 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 32512 bytes, then you could use `setString` for the input parameter and register the `OUT` parameter as `Types.VARCHAR` and use `getString` 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 wrapper 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 `setString` on the first parameter and `registerOutParameter` with `Types.CLOB` on the second.

Size Limitations

Please be aware of the effect on the performance of the Java memory management system due to creation of very large `byte` array or `String`. Please 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.

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. Inserting a temporary LOB may be preferable for some situations. For example, if the LOB data is relatively small so that 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(Connection, boolean, int)`. This method is defined in both the `oracle.sql.BLOB` and `oracle.sql.CLOB` classes. You free a temporary LOB with the `freeTemporary` method.

```
public static BLOB createTemporary(Connection conn, boolean isCached, int
duration);
public static CLOB createTemporary(Connection conn, boolean isCached, int
duration);
```

The duration must be either `DURATION_SESSION` or `DURATION_CALL` as defined in the `oracle.sql.BLOB` or `oracle.sql.CLOB` class. In client applications, `DURATION_SESSION` is appropriate. In Java stored procedures, you can use either `DURATION_SESSION` or `DURATION_CALL`, which ever is appropriate.

You can test whether a LOB is temporary 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. Otherwise, the storage used by the temporary LOB will not be reclaimed.

Notes:

- Failure to free a temporary LOB will result in the storage used by that LOB in the database being 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` instead of `getString` or `getBytes`. Also invoke `freeTemporary` to free the temporary LOBs.

Creating Temporary NCLOBs

You create temporary national character large objects (NCLOBs) using a variant of the `createTemporary` method.

```
CLOB.createTemporary (Connection conn, boolean cache, int duration, short form);
```

The `form` argument specifies whether the created LOB is a CLOB or an NCLOB. If `form` equals `oracle.jdbc.OraclePreparedStatement.FORM_NCHAR`, then the method creates an NCLOB. If `form` equals `oracle.jdbc.OraclePreparedStatement.FORM_CHAR`, then the method creates a CLOB.

Using Open and Close With LOBs

You do not have to open and close your LOBs. You may choose to open and close them for performance reasons.

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.

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 are done 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 see if a LOB is open or closed by calling the `isOpen` method. If you open the LOB by calling the `open(int)` method, 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`, 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.

Working with BFILEs

This section describes how to read and write data to and from BFILEs, using file locators. This section covers the following topics:

- Getting and Passing BFILE Locators
- Reading BFILE Data
- Creating and Populating a BFILE Column
- Accessing and Manipulating BFILE Data
- Additional BFILE Features

Getting and Passing BFILE Locators

Getter and setter methods are available for retrieving or passing BFILE locators from or to the database.

Retrieving BFILE Locators

Given a standard JDBC result set or callable statement object that includes BFILE locators, 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.

Notes:

- In the `OracleResultSet` and `OracleCallableStatement` classes, `getBFILE` and `getBfile` both return `oracle.sql.BFILE`. There is no `java.sql` interface for BFILEs.
- If using `getObject` or `getOracleObject`, remember to cast the output, as necessary.

Example: Getting a BFILE locator from a Result Set

Assume that the database has a table called `bfile_table` with a single column for the BFILE locator `bfile_col`. This example assumes that you have already created your `Statement` object `stmt`.

Select the BFILE locator into a standard result set. If you cast the result set to `OracleResultSet`, then you can use `getBFILE` to get the BFILE locator, as follows:

```
// Select the BFILE locator into a result set
ResultSet rs = stmt.executeQuery("SELECT bfile_col FROM bfile_table");
while (rs.next())
{
   oracle.sql.BFILE my_bfile = ((OracleResultSet)rs).getBFILE(1);
}
```

Note that as an alternative, you can use `getObject` to return the BFILE locator. In this case, because `getObject` returns a `java.lang.Object`, cast the results to `BFILE`. For example:

```
oracle.sql.BFILE my_bfile = (BFILE)rs.getObject(1);
```

Example: Getting a BFILE Locator from a Callable Statement

Assume you have an `OracleCallableStatement` object `ocs` that calls a function `func` that has a `BFILE` output parameter. The following code example sets up the callable statement, registers the output parameter as `OracleTypes.BFILE`, runs the statement, and retrieves the BFILE locator:

```
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("{? = call func()}");
ocs.registerOutParameter(1, OracleTypes.BFILE);
ocs.execute();
oracle.sql.BFILE bfile = ocs.getBFILE(1);
```

Passing BFILE Locators

To pass a BFILE locator to a prepared statement or callable statement, you can do 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.

Example: Passing a BFILE Locator to a Prepared Statement

Assume you want to insert a BFILE locator into a table, and you have an `OraclePreparedStatement` object `ops` to insert data into a table. The first column is a string, the second column is a BFILE, and you have a valid `oracle.sql.BFILE` object, `bfile`. Write the BFILE to the database, as follows:

```
OraclePreparedStatement ops = (OraclePreparedStatement)conn.prepareStatement
                           ("INSERT INTO my_bfile_table VALUES (?,?)");
ops.setString(1,"one");
ops.setBFILE(2, bfile);
ops.execute();
```

Example: Passing a BFILE Locator to a Callable Statement

Passing a BFILE locator to a callable statement is similar to passing it to a prepared statement. In this case, the BFILE locator is passed to the `myGetFileLength` procedure, which returns the BFILE length as a numeric value.

```
OracleCallableStatement cstmt = (OracleCallableStatement)conn.prepareCall
                               ("begin ? := myGetFileLength (?); end;");
try
{
   cstmt.registerOutParameter (1, Types.NUMERIC);
   cstmt.setBFILE (2, bfile);
   cstmt.execute ();
   return cstmt.getLong (1);
}
```

Reading BFILE Data

To read BFILE data, you must first get the BFILE locator. You can get the locator from either a callable statement or a result set. Once you obtain the locator, you can call a number of methods on the BFILE without opening it. For example, you can use the `oracle.sql.BFILE` methods `fileExists()` and `isFileOpen()` to determine whether the BFILE exists and if it is open. However, if you want to read and manipulate the data, then you must open and close the BFILE, as follows:

- Use the `openFile` method of the `oracle.sql.BFILE` class to open a BFILE.
- When you are done, use the `closeFile` method of the `BFILE` class.

BFILE data is through a Java stream. To read from a BFILE, use the `getBinaryStream` method of an `oracle.sql.BFILE` object to access the file as an input stream. This returns a `java.io.InputStream` object.

As with any `InputStream` object, use one of the overloaded `read` methods to read the file data and use the `close` method when you finish.

Notes:

- BFILEs are read-only. You cannot insert data or otherwise write to a BFILE.
- You can create a BFILE. However, you cannot create an operating system file that a BFILE would refer to. Those are created only externally.

Example: Reading BFILE Data

The following example uses the `getBinaryStream` method of an `oracle.sql.BFILE` object to read BFILE data into a byte stream and then read the byte stream into a byte array. The example assumes that the BFILE has already been opened.

```
// Read BFILE data from a BFILE locator
Inputstream in = bfile.getBinaryStream();
byte[] byte_array = new byte{10};
int byte_read = in.read(byte_array);
```

Creating and Populating a BFILE Column

This section discusses how to create a `BFILE` column in a table with SQL operations and specify the location where the BFILE resides. The examples in this section assume that you have already created your `Connection` object `conn` and `Statement` object `stmt`.

Creating a BFILE Column in a New Table

To work with BFILE data, create a `BFILE` column in a table, and specify the location of the BFILE. To specify the location of the BFILE, use the SQL `CREATE DIRECTORY...AS` statement to specify an alias for the directory where the BFILE resides. In this example, the directory alias is `test_dir` and the BFILE resides in the `/home/work` directory.

```
String cmd;
cmd = "CREATE DIRECTORY test_dir AS '/home/work'";
stmt.execute (cmd);
```

Use the SQL `CREATE TABLE` statement to create a table containing a `BFILE` column. In this example, the name of the table is `my_bfile_table`.

```
// Create a table containing a BFILE field
cmd = "CREATE TABLE my_bfile_table (x varchar2 (30), b bfile)";
stmt.execute (cmd);
```

In this example, the `VARCHAR2` column designates a row number and the `BFILE` column stores the locator of the BFILE data.

Populating a BFILE Column

Use the SQL `INSERT INTO...VALUES` statement to populate the `VARCHAR2` and `BFILE` fields. The `BFILE` column is populated with the locator to the BFILE data. To populate the `BFILE` column, use the `bfilename` function to specify the directory alias and the name of the BFILE file.

```
cmd ="INSERT INTO my_bfile_table VALUES ('one', bfilename(test_dir,
                                         'file1.data'))";
stmt.execute (cmd);
cmd ="INSERT INTO my_bfile_table VALUES ('two', bfilename(test_dir,
```

```
                                                'jdbcTest.data'))";
stmt.execute (cmd);
```

In this example, the name of the directory alias is `test_dir`. The locator of the BFILE `file1.data` is loaded into the `BFILE` column on row `one`, and the locator of the `BFILE` `jdbcTest.data` is loaded into the `bfile` column on row `two`.

As an alternative, you may want to create the row for the row number and BFILE locator now, but wait until later to insert the locator. In this case, insert the row number into the table and `null` as a place holder for the BFILE locator.

```
cmd ="INSERT INTO my_bfile_table VALUES ('three', null)";
stmt.execute(cmd);
```

Here, `three` is inserted into the row number column and `null` is inserted as the place holder. Later in your program, insert the BFILE locator into the table by using a prepared statement.

First get a valid BFILE locator into the `bfile` object:

```
rs = stmt.executeQuery("SELECT b FROM my_bfile_table WHERE x='two'");
rs.next();
oracle.sql.BFILE bfile = ((OracleResultSet)rs).getBFILE(1);
```

Then, create your prepared statement. Note that because this example uses the `setBFILE` method to identify the BFILE, the prepared statement must be cast to `OraclePreparedStatement`:

```
OraclePreparedStatement ops = (OraclePreparedStatement)conn.prepareStatement
                     (UPDATE my_bfile_table SET b=? WHERE x = 'three');
ops.setBFILE(1, bfile);
ops.execute();
```

Now row `two` and row `three` contain the same BFILE.

Once you have the BFILE locators available in a table, you can access and manipulate the BFILE data.

Accessing and Manipulating BFILE Data

Once you have the BFILE locator in a table, you can access and manipulate the data to which it points. To access and manipulate the data, you must first select its locator from a result set or a callable statement.

The following code continues the example from the preceding section, getting the locator of the BFILE from row `two` of a table into a result set. The result set is cast to `OracleResultSet` so that `oracle.sql.*` methods can be used on it. Several of the methods applied to the BFILE, such as `getDirAlias` and `getName`, do not require you to open the BFILE. Methods that manipulate the BFILE data, such as reading, getting the length, and displaying, *do* require you to open the BFILE.

When you finish manipulating the BFILE data, you must close the BFILE.

```
// select the bfile locator
cmd = "SELECT *  FROM my_bfile_table WHERE x = 'two'";
rset = stmt.executeQuery (cmd);

if (rset.next ())
    BFILE bfile = ((OracleResultSet)rset).getBFILE (2);

// for these methods, you do not have to open the bfile
println("getDirAlias() = " + bfile.getDirAlias());
```

```
println("getName() = " + bfile.getName());
println("fileExists() = " + bfile.fileExists());
println("isFileOpen() = " + bfile.isFileOpen());

// now open the bfile to get the data
bfile.openFile();

// get the BFILE data as a binary stream
InputStream in = bfile.getBinaryStream();
int length ;

// read the bfile data in 6-byte chunks
byte[] buf = new byte[6];

while ((length = in.read(buf)) != -1)
{
    // append and display the bfile data in 6-byte chunks
   StringBuffer sb = new StringBuffer(length);
   for (int i=0; i<length; i++)
      sb.append( (char)buf[i] );
   System.out.println(sb.toString());
}

// we are done working with the input stream. Close it.
in.close();

// we are done working with the BFILE. Close it.
bfile.closeFile();
```

Additional BFILE Features

In addition to the features already discussed in this chapter, the `oracle.sql.BFILE` class has a number of methods for further functionality, including the following:

- `closeFile`

  Closes the external file.

- `getBinaryStream`

  Returns the contents of the external file as a stream of bytes.

- `getBinaryStream(long)`

  Returns the contents of the external file as a stream of bytes beginning at the position in the external file specified by the argument.

- `getBytes`

  Reads from the external file, starting at a specified point, into a supplied buffer.

- `getName`

  Gets the name of the external file.

- `getDirAlias`

  Gets the directory alias of the external file.

- `isFileOpen`

  Determines whether the BFILE is open.

- `length`

Returns the length of the BFILE in bytes.

- `openFile`

  Opens the external file for read-only access.

- `position`

  Determines the byte position at which the given byte pattern begins.

17

Using Oracle Object References

This chapter describes Oracle extensions to standard Java Database Connectivity (JDBC) that let you access and manipulate object references. The following topics are discussed:

- Oracle Extensions for Object References
- Overview of Object Reference Functionality
- Retrieving and Passing an Object Reference
- Accessing and Updating Object Values through an Object Reference
- Custom Reference Classes with JPublisher

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 in Oracle JDBC, 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 weakly typed `oracle.sql.REF` instance, or 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.sql.ORAData` interface. The `oracle.sql.REF` class implements the standard `java.sql.Ref` 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.

Notes:

- If you are using the `oracle.sql.ORAData` interface for custom object classes, then you will presumably use `ORAData` 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.

Overview of Object Reference Functionality

To access and update object data through an object reference, you must obtain the reference instance through a result set or callable statement and then pass it back as a bind variable in a prepared statement or callable statement. It is the reference instance that contains the functionality to access and update object attributes.

This section covers the following topics:

- Object Reference Getter and Setter Methods
- Key REF Class Methods

Object Reference Getter and Setter Methods

You can use the result set, callable statement, and prepared statement methods to retrieve and pass object references.

Result Set and Callable Statement Getter Methods

The `OracleResultSet` and `OracleCallableStatement` classes support `getREF` and `getRef` methods to retrieve `REF` objects as output parameters. `REF` objects can be retrieved either as `oracle.sql.REF` instances or `java.sql.Ref` 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 `setREF` and `setRef` methods to take `REF` 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.REF` instance or a `java.sql.Ref` instance.

Key REF Class Methods

You can use the following `oracle.sql.REF` class methods to retrieve the SQL object type name and retrieve and pass the underlying object data:

- `getBaseTypeName`

Retrieves the fully-qualified SQL structured type name of the referenced object. This is a standard method specified by the `java.sql.Ref` interface.

- `getValue`

  Retrieves the referenced object from the database, enabling you to access its attribute values. It optionally takes a type map object. You can use the default type map of the database connection object. This method is an Oracle extension.

- `setValue`

  Sets the referenced object in the database, allowing you to update its attribute values. It takes an instance of the object type, either a `STRUCT` instance or an instance of a custom object class, as input. This method is an Oracle extension.

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 a `REF` object.
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 `getValue` 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())
{
   REF ref = ((OracleResultSet)rs).getREF(1);
   Address a = (Address)
ref.getValue();
}
```

> **Note:** In the preceding code, `stmt` is a previously defined statement object.

As with other SQL types, you could retrieve the reference with the `getObject` method of your result set. Note that this would require you to cast the output. For example:

```
REF ref = (REF)rs.getObject(1);
```

There are no performance advantages in using `getObject` instead of `getREF`; however, using `getREF` enables you to avoid casting the output.

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 = ?");
((OraclePreparedStatement)pstmt).setREF (1, addr_ref);
((OraclePreparedStatement)pstmt).setROWID (2, rowid);
```

Accessing and Updating Object Values through an Object Reference

You can use the REF object setValue 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 17-3, 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.getValue();
}
```

Then, you can create a Java Address object that corresponds to the database ADDRESS object. Use the setValue method of the REF class to set the value of the database object, as follows:

```
Address addr = new Address(...);
ref.setValue(addr);
```

Here, the setValue method updates the database ADDRESS object immediately.

Custom Reference Classes with JPublisher

This chapter primarily describes the functionality of the oracle.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.sql.ORAData 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.sql.ORADataFactory 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.

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 a ORAData implementation, then JPublisher will also generate a custom reference class that implements ORAData and ORADataFactory and includes an oracle.sql.REF attribute. The ORAData 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 15-32
- *Oracle Database JPublisher User's Guide*

18

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:

- Oracle Extensions for Collections
- Overview of Collection Functionality
- ARRAY Performance Extension Methods
- Creating and Using Arrays
- Using a Type Map to Map Array Elements
- Custom Collection Classes with JPublisher

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
- Creating Collections
- Creating Multilevel Collection Types

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.sql.ORAData` interface. In addition, the custom class or a companion class must implement `oracle.sql.ORADataFactory`. 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 18-18.

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 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.

Note: Multilevel collection types are available only for Oracle9*i* and later.

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 the following topics:

- Array Getter and Setter Methods
- ARRAY Descriptors and ARRAY Class Functionality

Array Getter and Setter Methods

Use the following result set, callable statement, and prepared statement methods to retrieve and pass collections as Java arrays.

Result Set and Callable Statement Getter Methods

The `OracleResultSet` and `OracleCallableStatement` classes 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.

Note: The 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 an array type in the database, the cached descriptor for that array type will become stale and your application will receive a `SQLException`.

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 Descriptors and ARRAY Class Functionality

The section introduces `ARRAY` descriptors and lists methods of the `ARRAY` class to provide an overview of its functionality.

ARRAY Descriptors

Creating and using an `ARRAY` object requires the existence of a descriptor, that is, an instance of the `oracle.sql.ArrayDescriptor` class, to exist for the SQL type of the collection being materialized in the array. You need only one `ArrayDescriptor` object for any number of `ARRAY` objects that correspond to the same SQL type.

See Also: "Creating ARRAY Objects and Descriptors" on page 18-7

ARRAY Class Methods

The `oracle.sql.ARRAY` class includes the following methods:

- `getDescriptor`

  Returns the `ArrayDescriptor` object that describes the array type.

- `getArray`

  Retrieves the contents of the array in default JDBC types. If it retrieves an array of objects, then `getArray` uses the default type map of the database connection object to determine the types.

- `getOracleArray`

  Is identical to `getArray`, but retrieves the elements in `oracle.sql.*` format.

- `getBaseType`

  Returns the SQL type code for the array elements.

- `getBaseTypeName`

  Returns the SQL type name of the elements of this array.

- `getSQLTypeName`

  Returns the fully qualified SQL type name of the array as a whole. This method is an Oracle extension.

- `getResultSet`

  Materializes the array elements as a result set.

- `getJavaSQLConnection`

  Returns the connection instance associated with this array.

- `length`

  Returns the number of elements in the array.

> **Note:** As an example of the difference between `getBaseTypeName` and `getSQLTypeName`, if you define `ARRAY_OF_PERSON` as the array type for an array of `PERSON` objects in the `SCOTT` schema, then `getBaseTypeName` would return `SCOTT.PERSON` and `getSQLTypeName` would return `SCOTT.ARRAY_OF_PERSON`.

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 allow 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.

Here are the methods:

- ```
  public int[] getIntArray() throws SQLException

  public int[] getIntArray(long index, int count)
     throws SQLException
  ```

- ```
  public long[] getLongArray() throws SQLException

  public long[] getLongArray(long index, int count)
     throws SQLException
  ```

- ```
  public float[] getFloatArray() throws SQLException

  public float[] getFloatArray(long index, int count)
     throws SQLException
  ```

- ```
  public double[] getDoubleArray() throws SQLException

  public double[] getDoubleArray(long index, int count)
     throws SQLException
  ```

- ```
  public short[] getShortArray() throws SQLException

  public short[] getShortArray(long index, int count)
     throws SQLException
  ```
  ```

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

The 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:

- public void setAutoBuffering(boolean enable)
- public boolean getAutoBuffering()

The setAutoBuffering 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 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:

- public synchronized void setAutoIndexing(boolean enable, int direction) throws SQLException
- public synchronized void setAutoIndexing(boolean enable) throws SQLException

The setAutoIndexing method sets the auto-indexing mode for the oracle.sql.ARRAY object. The direction parameter gives the array object a hint. Specify this parameter to help the JDBC driver determine the best indexing scheme. The following are the values you can specify for the direction parameter:

- ARRAY.ACCESS_FORWARD
- ARRAY.ACCESS_REVERSE
- ARRAY.ACCESS_UNKNOWN

The setAutoIndexing(boolean) method signature sets the access direction as ARRAY.ACCESS_UNKNOWN by default.

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 and Descriptors
- Retrieving an Array and Its Elements
- Passing Arrays to Statement Objects

Creating ARRAY Objects and Descriptors

This section describes how to create ARRAY objects and descriptors and lists useful methods of the ArrayDescriptor class. This section covers the following topics:

Steps in Creating ArrayDescriptor and ARRAY Objects

This section describes how to construct an oracle.sql.ARRAY object. To do this, you must:

1. Create an ArrayDescriptor object for the array, if one does not already exist.
2. Use the ArrayDescriptor object to construct the oracle.sql.ARRAY object for the array you want to pass.

An ArrayDescriptor is an object of the oracle.sql.ArrayDescriptor class and describes the SQL type of an array. Only one array descriptor is necessary for any one SQL type. The driver caches ArrayDescriptor objects to avoid re-creating them if the SQL type has already been encountered. You can reuse the same descriptor object to create multiple instances of an oracle.sql.ARRAY object for the same array type.

Collections are strongly typed. Oracle supports only named collections, that is, a collection given a SQL type name. For example, you create a collection with the CREATE TYPE statement, as follows:

```
CREATE TYPE num_varray AS varray(22) OF NUMBER(5,2);
```

num_varray is the SQL type name for the collection type.

Note: The name of the collection type is not the same as the type name of the elements. For example:

```
CREATE TYPE person AS object
           (c1 NUMBER(5), c2 VARCHAR2(30));
CREATE TYPE array_of_persons AS varray(10)
           OF person;
```

In the preceding statements, the SQL name of the collection type is ARRAY_OF_PERSON. The SQL name of the collection elements is PERSON.

Before you can construct an Array object, an ArrayDescriptor must first exist for the given SQL type of the array. If an ArrayDescriptor does not exist, then you must construct one by passing the SQL type name of the collection type and your

Connection object, which JDBC uses to connect to the database to gather meta data, to the constructor.

```
ArrayDescriptor arraydesc = ArrayDescriptor.createDescriptor
                                          (sql_type_name, connection);
```

sql_type_name is the type name of the array and *connection* is your Connection object.

Once you have your ArrayDescriptor object for the SQL type of the array, you can construct the ARRAY object. To do this, pass in the array descriptor, your connection object, and a Java object containing the individual elements you want the array to contain.

```
ARRAY array = new ARRAY(arraydesc, connection, elements);
```

arraydesc is the array descriptor created previously, connection is your connection object, and elements is a Java array. 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[].

> **Note:** The setARRAY, setArray, and setObject methods of the OraclePreparedStatement class take an object of the type oracle.sql.ARRAY as an argument, not an array of objects.

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 oracle.sql.ARRAY constructor is defined as follows:

```
public ARRAY(ArrayDescriptor type, Connection conn, Object elements)
throws SQLException
```

The first argument is an oracle.sql.ArrayDescriptor object that describes the multilevel collection type. The second argument is the current database connection. And the third argument is a java.lang.Object that holds the multilevel collection elements. This is the same constructor used to create single-level collections, but can create multilevel collections as well. The elements parameter can now be either a one dimension array or a nested Java array.

To create a single-level collection, the elements are a one dimensional Java array. 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:

```
Connection conn = ...;          // make a JDBC connection

// create the collection types
Statement stmt = conn.createStatement ();
stmt.execute ("CREATE TYPE varray1 AS VARRAY(10) OF NUMBER(12, 2)"); // one
                                                                     // layer
stmt.execute ("CREATE TYPE varray2 AS VARRAY(10) OF varray1"); // two layers
stmt.execute ("CREATE TYPE varray3 AS VARRAY(10) OF varray2"); // three layers
stmt.execute ("CREATE TABLE tab2 (col1 index, col2 value)");
stmt.close ();
```

```
// obtain a type descriptor of "SCOTT.VARRAY3"
ArrayDescriptor desc = ArrayDescriptor.createDescriptor("SCOTT.VARRAY3", conn);

// prepare the multi level collection elements as a nested Java array
int[][][] elems = { {{1}, {1, 2}}, {{2}, {2, 3}}, {{3}, {3, 4}} };

// create the ARRAY by calling the constructor
ARRAY array3 = new ARRAY (desc, conn, elems);

// some operations
...

// close the database connection
conn.close();
```

In the preceding example, another implementation is to prepare the `elems` as a Java array of `oracle.sql.ARRAY[]` elements, and each `oracle.sql.ARRAY[]` element represents a `SCOTT.VARRAY3`.

Using ArrayDescriptor Methods

An `ARRAY` descriptor can be referred to as a type object. It has information about the SQL name of the underlying collection, the type code of the array elements, and if it is an array of structured objects, then the SQL name of the elements. The descriptor also contains the information about converting to and from the given type. You need only one descriptor object for any one type, then you can use that descriptor to create as many arrays of that type as you want.

The `ArrayDescriptor` class has the following methods for retrieving the type code and type name of an element:

- `createDescriptor`

  Is a factory for `ArrayDescriptor` instances. It looks up the name in the database and determine the characteristics of the array.

- `getBaseType`

  Returns the integer type code associated with this `ARRAY` descriptor.

- `getBaseName`

  Returns a string with the type name associated with this array element if it is a `STRUCT` or `REF`.

- `getArrayType`

  Returns an integer indicating whether the array is a VARRAY or nested table. `ArrayDescriptor.TYPE_VARRAY` and `ArrayDescriptor.TYPE_NESTED_TABLE` are the possible return values.

- `getMaxLength`

  Returns the maximum number of elements for this array type.

- `getJavaSQLConnection`

  Returns the connection instance that was used in creating the `ARRAY` descriptor.

> **Note:** In releases prior to Oracle9*i* Database, you could not use a collection within a collection. You could, however, use a structured object with a collection attribute, or a collection with structured object elements. In Oracle9*i* Database and later releases, you can use a collection within a collection.

Serializable ARRAY Descriptors

When you create an `ARRAY` object, you first must create an `ArrayDescriptor` object. Create the `ArrayDescriptor` object by calling the `ArrayDescriptor.createDescriptor` method. The `oracle.sql.ArrayDescriptor` class is serializable, meaning that you can write the state of an `ArrayDescriptor` object to an output stream for later use. Re-create the `ArrayDescriptor` object by reading its serialized state from an input stream. This is referred to as deserializing. With the `ArrayDescriptor` object serialized, you do not need to call the `createDescriptor` method, simply deserialize the `ArrayDescriptor` object.

It is advisable to serialize an `ArrayDescriptor` object when the object type is complex but not changed often.

If you create an `ArrayDescriptor` object through deserialization, then you must provide the appropriate database connection instance for the `ArrayDescriptor` object using the `setConnection` method.

The following code furnishes the connection instance for an `ArrayDescriptor` object:

```
public void setConnection (Connection conn) throws SQLException
```

> **Note:** The JDBC driver does not verify that the connection object from the `setConnection` method connects to the same database from which the type descriptor was initially derived.

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.sql.STRUCT` 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 the 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.sql.STRUCT` object.

The `getResultSet(`*`map`*`)` method behaves similarly to the `getArray(`*`map`*`)` method.

See Also: "Using a Type Map to Map Array Elements" on page 18-16

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 will be of the `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`; `getARRAY` 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. Construct an `ArrayDescriptor` object for the SQL type that the array will contain, as follows:

```
ArrayDescriptor descriptor = ArrayDescriptor.createDescriptor
                                    (sql_type_name, connection);
```

sql_type_name is a Java string specifying the user-defined SQL type name of the array and *connection* is your `Connection` object.

2. Define the array that you want to pass to the prepared statement as an `oracle.sql.ARRAY` object.

```
ARRAY array = new ARRAY(descriptor, connection, elements);
```

descriptor is the `ArrayDescriptor` object previously constructed and *elements* is a `java.lang.Object` containing a Java array of the elements.

3. Create a `java.sql.PreparedStatement` object containing the SQL statement to be run.

4. 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.

5. 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.sql.STRUCT` 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('Scott Tiger', 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.sql.ORAData` 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.sql.ORADataFactory` 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 `ORAData` and `ORADataFactory` 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 `ORAData` 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 15-32
- *Oracle Database JPublisher User's Guide*

19

Result Set

Standard Java Database Connectivity (JDBC) 2.0 features in Java Development Kit (JDK) 1.2.x 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 these features, including the following topics:

- Overview
- Creating Scrollable or Updatable Result Sets
- Positioning and Processing in Scrollable Result Sets
- Updating Result Sets
- Fetch Size
- Refetching Rows
- Seeing Database Changes Made Internally and Externally
- Summary of New Methods for Result Set Enhancements

Overview

This section provides an overview of JDBC 2.0 result set functionality and categories, and some discussion of implementation requirements for the Oracle JDBC drivers. This section covers the following topics:

- Result Set Functionality and Result Set Categories Supported in JDBC 2.0
- Oracle JDBC Implementation Overview for Result Set Enhancements

Result Set Functionality and Result Set Categories Supported in JDBC 2.0

Result set functionality in JDBC 2.0 includes enhancements for scrollability and positioning, sensitivity to changes by others, and updatability.

- Scrollability, positioning, and sensitivity are determined by the result set type.
- Updatability is determined by the concurrency type.

Specify the desired result set type and concurrency type when you create the statement object that will produce the result set.

Together, the various result set types and concurrency types provide for six different categories of result set.

Scrollability, Positioning, and Sensitivity

Scrollability refers to the ability to move backward as well as forward through a result set. Associated with scrollability is the ability to move to any particular position in the result set, through either relative positioning or absolute positioning.

Relative positioning enables you to move a specified number of rows forward or backward from the current row. Absolute positioning enables you to move to a specified row number, counting from either the beginning or the end of the result set.

Under JDBC 2.0, scrollable/positionable result sets are also available.

When creating a scrollable/positionable result set, you must also specify sensitivity. This refers to the ability of a result set to detect and reveal changes made to the underlying database from outside the result set.

A sensitive result set can see changes made to the database while the result set is open, providing a dynamic view of the underlying data. Changes made to the underlying columns values of rows in the result set are visible.

An insensitive result set is *not* sensitive to changes made to the database while the result set is open, providing a static view of the underlying data. You would need to retrieve a new result set to see changes made to the database.

Result Set Types for Scrollability and Sensitivity

When you create a result set under JDBC 2.0 functionality, you must choose a particular result set type to specify whether the result set is scrollable/positional and sensitive to underlying database changes.

If the JDBC 1.0 functionality is all you desire, JDBC 2.0 continues to support this through the forward-only result set type. A forward-only result set cannot be sensitive.

If you want a scrollable result set, then you must also specify sensitivity. Specify the scroll-sensitive type for the result set to be scrollable and sensitive to underlying changes. Specify the scroll-insensitive type for the result set to be scrollable but not sensitive to underlying changes.

To summarize, the following result set types are available with JDBC 2.0:

- Forward-only

  This is a JDBC 1.0 functionality. This type of result set is not scrollable, not positionable, and not sensitive.

- Scroll-sensitive

  This type of result set is scrollable and positionable. It is also sensitive to underlying database changes.

- Scroll-insensitive

  This type of result set is scrollable and positionable, but not sensitive to underlying database changes.

> **Note:** The sensitivity of a scroll-sensitive result set is affected by fetch size.

Updatability

Updatability refers to the ability to update data in a result set and then copy the changes to the database. This includes inserting new rows into the result set or deleting existing rows.

Updatability might also require database write locks to mediate access to the underlying database. Because you cannot have multiple write locks concurrently, updatability in a result set is associated with concurrency in database access.

Result sets can optionally be updatable under JDBC 2.0

Note: Updatability is independent of scrollability and sensitivity. Although, it is typical for an updatable result set to also be scrollable so that you can position it to particular rows that you want to update or delete.

Concurrency Types for Updatability

The concurrency type of a result set determines whether it is updatable. Under JDBC 2.0, the following concurrency types are available:

- Updatable

  In this case, updates, inserts, and deletes can be performed on the result set and copied to the database.

- Read-only

  The result set cannot be modified in any way.

Summary of Result Set Categories

Because scrollability and sensitivity are independent of updatability, the three result set types and two concurrency types combine for a total of six result set categories, as follows:

- forward-only/read-only
- forward-only/updatable
- scroll-sensitive/read-only
- scroll-sensitive/updatable
- scroll-insensitive/read-only
- scroll-insensitive/updatable

Note: A forward-only updatable result set has no positioning functionality. You can only update rows as you iterate through them with the `next` method.

Oracle JDBC Implementation Overview for Result Set Enhancements

This section discusses key aspects of the Oracle JDBC implementation of result set enhancements 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, the 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.

Implementing a Custom Client-Side Cache for Scrollability

There is some flexibility in how to implement client-side caching in support of JDBC 2.0 scrollable result sets.

Although Oracle JDBC provides a complete implementation, it also supplies an interface, `OracleResultSetCache`, that you can implement as desired:

```
public interface OracleResultSetCache
{
  /**
   * Save the data in the i-th row and j-th column.
   */
  public void put (int i, int j, Object value) throws IOException;

  /**
   * Return the data stored in the i-th row and j-th column.
   */
  public Object get (int i, int j) throws IOException;

  /**
   * Remove the i-th row.
   */
  public void remove (int i) throws IOException;

  /**
   * Remove the data stored in i-th row and j-th column
   */
  public void remove (int i, int j) throws IOException;

  /**
   * Remove all data from the cache.
   */
  public void clear () throws IOException;
```

```
  /**
   * Close the cache.
   */
  public void close () throws IOException;
}
```

If you implement this interface with your own class, then your application code must instantiate your class and then use the `setResultSetCache` method of an `OracleStatement`, `OraclePreparedStatement`, or `OracleCallableStatement` object to set the caching mechanism to use your implementation. Following is the method signature:

```
void setResultSetCache(OracleResultSetCache cache) throws SQLException
```

Call this method prior to running a query. The result set produced by the query will then use your specified caching mechanism.

Creating Scrollable or Updatable Result Sets

In using JDBC 2.0 result set enhancements, you may specify the result set type and the concurrency type when you create a generic statement or prepare a prepared statement or callable statement that will run a query.

This section discusses the creation of result sets to use JDBC 2.0 enhancements. It covers the following topics:

- Specifying Result Set Scrollability and Updatability
- Result Set Limitations and Downgrade Rules

Specifying Result Set Scrollability and Updatability

Under JDBC 2.0, the `Connection` class has the following methods that take a result set type and a concurrency type as input:

- `Statement createStatement(int resultSetType, int resultSetConcurrency)`
- `PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency)`
- `CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency)`

The statement objects created will have the intelligence to produce the appropriate kind of result sets.

You can specify one of the following `static` constant values for result set type:

- `ResultSet.TYPE_FORWARD_ONLY`
- `ResultSet.TYPE_SCROLL_INSENSITIVE`
- `ResultSet.TYPE_SCROLL_SENSITIVE`

> **Tip:** "Oracle Implementation of Scroll-Sensitive Result Sets" on page 19-19

And you can specify one of the following `static` constant values for concurrency type:

- `ResultSet.CONCUR_READ_ONLY`

- `ResultSet.CONCUR_UPDATABLE`

After creating a `Statement`, `PreparedStatement`, or `CallableStatement` object, you can verify its result set type and concurrency type by calling the following methods on the statement object:

- `int getResultSetType() throws SQLException`
- `int getResultSetConcurrency() throws SQLException`

Example 19–1 Prepared Statement Object With Result Set

Following is an example of a prepared statement object that specifies a scroll-sensitive and updatable result set for queries run through that statement:

```
...
PreparedStatement pstmt = conn.prepareStatement
  ("SELECT empno, sal FROM emp WHERE empno = ?",
  ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

pstmt.setString(1, "28959");
ResultSet rs = pstmt.executeQuery();
...
```

Result Set 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 will result 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.

> **See Also:** "Summary of New Methods for Result Set Enhancements" on page 19-20

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 the 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`.

> **Notes:** 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.

Positioning and Processing in Scrollable Result Sets

Scrollable result sets enable you to iterate through them, either forward or backward, and to position the result set to any desired row.

This section discusses positioning within a scrollable result set and how to process a scrollable result set backward, instead of forward. It covers the following sections:

Positioning in a Scrollable Result Set

In a scrollable result set, you can use several result set methods to move to a desired position and to check the current position.

Note: You cannot position a forward-only result set. Any attempt to position it or to determine the current position will result in a `SQLException`.

Methods for Moving to a New Position

The following result set methods are available for moving to a new position in a scrollable result set:

- `void beforeFirst() throws SQLException`

  Positions to before the first row of the result set, or has no effect if there are no rows in the result set. This is where you would typically start iterating through a result set to process it going forward and is the default initial position for any kind of result set.

  You are outside the result set bounds after a `beforeFirst()` call. There is no valid current row, and you cannot position relatively from this point.

- `void afterLast() throws SQLException`

  Positions to after the last row of the result set, or has no effect if there are no rows in the result set. This is where you would typically start iterating through a result set to process it going backward.

  You are outside the result set bounds after an `afterLast()` call. There is no valid current row, and you cannot position relatively from this point.

- `boolean first() throws SQLException`

  Positions to the first row of the result set, or returns `false` if there are no rows in the result set.

- `boolean last() throws SQLException`

  Positions to the last row of the result set, or returns `false` if there are no rows in the result set.

- `boolean absolute(int row) throws SQLException`

  Positions to an absolute row from either the beginning or end of the result set. If you input a positive number, then it positions from the beginning. If you input a negative number, then it positions from the end. This method returns `false` if there are no rows in the result set.

  Attempting to move forward beyond the last row, such as an `absolute(11)` call if there are 10 rows, will position to after the last row, having the same effect as an `afterLast()` call.

Attempting to move backward beyond the first row, such as an `absolute(-11)` call if there are 10 rows, will position to before the first row, having the same effect as a `beforeFirst()` call.

> **Note:** Calling `absolute(1)` is equivalent to calling `first()`; calling `absolute(-1)` is equivalent to calling `last()`.

- `boolean relative(int row) throws SQLException`

  Moves to a position relative to the current row, forward if you input a positive number or backward if you input a negative number, or returns `false` if there are no rows in the result set.

  The result set must be at a valid current row for use of the `relative` method.

  Attempting to move forward beyond the last row will position to after the last row, having the same effect as an `afterLast()` call.

  Attempting to move backward beyond the first row will position to before the first row, having the same effect as a `beforeFirst()` call.

  A `relative(0)` call is valid but has no effect.

> **Note:** You cannot position relatively from before the first row, which is the default initial position, or after the last row. Attempting relative positioning from either of these positions would result in a `SQLException`.

Methods for Checking the Current Position

The following result set methods are available for checking the current position in a scrollable result set:

- `boolean isBeforeFirst() throws SQLException`

  Returns `true` if the position is before the first row.

- `boolean isAfterLast() throws SQLException`

  Returns `true` if the position is after the last row.

- `boolean isFirst() throws SQLException`

  Returns `true` if the position is at the first row.

- `boolean isLast() throws SQLException`

  Returns `true` if the position is at the last row.

- `int getRow() throws SQLException`

  Returns the row number of the current row, or returns 0 if there is no valid current row.

> **Note:** The `boolean` methods, `isFirst()`, `isLast()`, `isAfterFirst()`, and `isAfterLast()`, all return `false`. Also, they do *not* throw an exception if there are no rows in the result set.

Processing a Scrollable Result Set

In a scrollable result set you can iterate backward instead of forward as you process the result set. The following methods are available:

- `boolean next() throws SQLException`
- `boolean previous() throws SQLException`

The `previous()` method works similarly to the `next()` method, in that it returns `true` as long as the new current row is valid, and `false` as soon as it runs out of rows, that is, has passed the first row.

Backward versus Forward Processing

You can process the entire result set going forward, using the `next()` method. The default initial position in the result set is before the first row, appropriately, but you can call the `beforeFirst()` method if you have moved elsewhere since the result set was created.

To process the entire result set going backward, call `afterLast()`, then use the `previous()` method. For example:

```
...
/* NOTE: The specified concurrency type, CONCUR_UPDATABLE, is not relevant to this
example. */

Statement stmt = conn.createStatement
         (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");

rs.afterLast();
while (rs.previous())
{
   System.out.println(rs.getString("empno") + " " + rs.getFloat("sal"));
}
...
```

Unlike relative positioning, you can use `next()` from before the first row and `previous()` from after the last row. You do not have to be at a valid current row to use these methods.

> **Note:** In a non-scrollable result set, you can process only with the `next()` method. Attempting to use the `previous()` method will cause a `SQLException`.

Presetting the Fetch Direction

The JDBC 2.0 standard allows the ability 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`

The Oracle JDBC drivers support only the forward preset value, which you can specify by inputting 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.

Updating Result Sets

A concurrency type of `CONCUR_UPDATABLE` enables you to update rows in the result set, delete rows from the result set, or insert rows into the result set.

After you perform an `UPDATE` or `INSERT` operation in a result set, you propagate the changes to the database in a separate step that you can skip if you want to cancel the changes.

However, a `DELETE` operation in a result set is immediately run, but not necessarily committed, in the database as well.

Note: When using an updatable result set, it is typical to also make it scrollable. This enables you to position to any row that you want to change. With a forward-only updatable result set, you can change rows only as you iterate through them with the `next()` method.

This section covers the following topics:

- Performing a DELETE Operation in a Result Set
- Performing an UPDATE Operation in a Result Set
- Performing an INSERT Operation in a Result Set
- Avoiding Update Conflicts

Performing a DELETE Operation in a Result Set

The result set `deleteRow()` method will delete the current row. Following is the method signature:

```
void deleteRow() throws SQLException
```

Note: Unlike `UPDATE` and `INSERT` operations in a result set, which require a separate step to propagate the changes to the database, a `DELETE` operation in a result set is immediately run in the corresponding row in the database as well.

Once you call `deleteRow()`, the changes will be made permanent with the next transaction `COMMIT` operation. Remember also that by default, the auto-commit flag is set to `true`. Therefore, unless you override this default, any `deleteRow()` operation will be run and committed immediately.

Presuming the result set is also scrollable, you can position to a row using any of the available positioning methods, except `beforeFirst()` and `afterLast()`, which do not go to a valid current row, and then delete that row, as in the following example:

```
...
rs.absolute(5);
```

```
rs.deleteRow();
...
```

Important: The deleted row remains in the result set object even after it has been deleted from the database.

In a scrollable result set, by contrast, a `DELETE` operation is evident in the local result set object. The row would no longer be in the result set after the `DELETE`. The row preceding the deleted row becomes the current row, and row numbers of subsequent rows are changed accordingly.

Performing an UPDATE Operation in a Result Set

Performing a result set `UPDATE` operation requires two separate steps to first update the data in the result set and then copy the changes to the database.

Presuming the result set is also scrollable, you can position to a row using any of the available positioning methods, except `beforeFirst()` and `afterLast()`, which do not go to a valid current row, and then update that row as desired.

Here are the steps for updating a row in the result set and database:

1. Call the appropriate `update`*`XXX`* methods to update the data in the columns you want to change.

   With JDBC 2.0, a result set object has an `update`*`XXX`* method for each data type, as with the `set`*`XXX`* methods previously available for updating the database directly.

   Each of these methods takes an `int` for the column number or a string for the column name and then an item of the appropriate data type to set the new value. Following are a couple of examples for a result set `rs`:

   ```
   rs.updateString(1, "mystring");
   rs.updateFloat(2, 10000.0f);
   ```

2. Call the `updateRow` method to copy the changes to the database or the `cancelRowUpdates` method to cancel the changes.

   Once you call `updateRow`, the changes are run and will be made permanent with the next transaction `COMMIT` operation. Be aware that by default, the auto-commit flag is set to `true` so that any operation run is committed immediately.

   If you choose to cancel the changes before copying them to the database, then call the `cancelRowUpdates` method instead. This will also revert to the original values for that row in the local result set object. Note that once you call the `updateRow` method, the changes are written to the transaction and cannot be canceled unless you roll back the transaction.

   Note: Auto-commit must be disabled to allow a `ROLLBACK` operation.

   Positioning to a different row before calling `updateRow` also cancels the changes and reverts to the original values in the result set.

   Before calling `updateRow`, you can call the usual `get`*`XXX`* methods to verify that the values have been updated correctly. These methods take an `int` column index or string column name as input. For example:

```
float myfloat = rs.getFloat(2);
...
// process myfloat to see if it's appropriate
...
```

> **Note:** Result set `UPDATE` operations are visible in the local result set object for all result set types, forward-only, scroll-sensitive, and scroll-insensitive.

Example

Following is an example of a result set `UPDATE` operation that is also copied to the database. The tenth row is updated. The column number is used to specify column 1, and the column name, `sal`, is used to specify column 2.

```
Statement stmt = conn.createStatement
         (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");
if (rs.absolute(10))        // (returns false if row does not exist)
{
   rs.updateString(1, "28959");
   rs.updateFloat("sal", 100000.0f);
   rs.updateRow();
}
// Changes are made permanent with the next COMMIT operation.
```

Performing an INSERT Operation in a Result Set

Result set `INSERT` operations use what is called the result set insert-row, which is a staging area that holds the data for the inserted row until it is copied to the database. You must explicitly move to this row to write the data that will be inserted.

As with `UPDATE` operations, result set `INSERT` operations require separate steps to first write the data to the insert-row and then copy it to the database.

Following are the steps in running a result set `INSERT` operation.

1. Move to the insert-row by calling the result set `moveToInsertRow` method.

   > **Note:** The result set will remember the current position prior to the `moveToInsertRow` call. Afterward, you can go back to it with a `moveToCurrentRow` call.

2. As with `UPDATE` operations, use the appropriate `update`*`XXX`* methods to write data to the columns. For example:

   ```
   rs.updateString(1, "mystring");
   rs.updateFloat(2, 10000.0f);
   ```

   You can specify a string for column name, instead of an integer for column number.

> **Important:** Each column value in the insert-row is undefined until you call the `updateXXX` method for that column. You must call this method and specify a non-null value for all non-nullable columns, or else attempting to copy the row into the database will result in a `SQLException`.
>
> However, it is permissible to *not* call `updateXXX` for a nullable column. This will result in a value of `null`.

3. Copy the changes to the database by calling the result set `insertRow` method.

   Once you call `insertRow`, the insert is processed and will be made permanent with the next transaction `COMMIT` operation.

   Positioning to a different row before calling `insertRow` cancels the insert and clears the insert-row.

   Before calling `insertRow` you can call the usual `getXXX` methods to verify that the values have been set correctly in the insert-row. These methods take an `int` column index or string column name as input. For example:

```
float myfloat = rs.getFloat(2);
...process myfloat to see if it's appropriate...
```

> **Note:** No result set type can see a row inserted by a result set `INSERT` operation.

Example

The following example performs a result set `INSERT` operation, moving to the insert-row, writing the data, copying the data into the database, and then returning to what was the current row prior to going to the insert-row. The column number is used to specify column 1, and the column name, `sal`, is used to specify column 2.

```
...
Statement stmt = conn.createStatement
         (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");

rs.moveToInsertRow();
rs.updateString(1, "28959");
rs.updateFloat("sal", 100000.0f);
rs.insertRow();
// Changes will be made permanent with the next COMMIT operation.
rs.moveToCurrentRow();  // Go back to where we came from...
...
```

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.

The 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-prefetch value. You can change the number of rows retrieved with each trip to the database cursor by changing the row-prefetch value.

JDBC 2.0 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.

This section covers the following topics:

- Setting the Fetch Size
- Use of Standard Fetch Size versus Oracle Row-Prefetch Setting

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.

Use of Standard Fetch Size versus Oracle Row-Prefetch Setting

Using the JDBC 2.0 fetch size is fundamentally similar to using the Oracle row-prefetch value, except that with the row-prefetch value you do not have the flexibility of distinct values in the statement object and result set object. The row prefetch value would be used everywhere.

Furthermore, JDBC 2.0 fetch size usage is portable and can be used with other JDBC drivers. Oracle row-prefetch usage is vendor-specific.

See Also: "Oracle Row Prefetching" on page 25-15

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.

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.

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`.

Seeing Database Changes Made Internally and Externally

This section discusses the ability of a result set to see the following:

- Its own changes, 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 Sun Microsystems JDBC 2.0 specification.

This section covers the following topics:

- Seeing Internal Changes
- Seeing External Changes
- Visibility versus Detection of External Changes
- Summary of Visibility of Internal and External Changes
- Oracle Implementation of Scroll-Sensitive Result Sets

Seeing Internal Changes

The ability of an updatable result set to see its own changes depends on both the result set type and the kind of change. This is summarized as follows:

- Internal `DELETE` operations are visible for scrollable result sets, but are not visible for forward-only result sets.

  After you delete a row in a scrollable result set, the preceding row becomes the new current row, and subsequent row numbers are updated accordingly.
- Internal `UPDATE` operations are always visible, regardless of the result set type.
- Internal `INSERT` operations are never visible, regardless of the result set type.

An internal change being visible essentially means that a subsequent `get`*`XXX`* call will see the data changed by a preceding `update`*`XXX`* call on the same data item.

JDBC 2.0 `DatabaseMetaData` objects include the following methods to verify this:

- `boolean ownDeletesAreVisible(int) throws SQLException`
- `boolean ownUpdatesAreVisible(int) throws SQLException`
- `boolean ownInsertsAreVisible(int) throws SQLException`

Each takes a result set type, `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_SENSITIVE`, or `ResultSet.TYPE_SCROLL_INSENSITIVE`, as input.

> **Note:** When you make an internal change that causes a trigger to run, the trigger changes are effectively external changes. However, if the trigger affects data in the row you are updating, then you *will* see those changes for any scrollable/updatable result set, because an implicit row refetch occurs after the update.

Seeing External Changes

Only a scroll-sensitive result set can see external changes to the underlying database, and it can only see the changes from external `UPDATE` operations. Changes from external `DELETE or INSERT` operations are never visible.

> **Note:** Any discussion of seeing changes from outside the enclosing transaction presumes the transaction itself has an isolation level setting that allows the changes to be visible.

JDBC 2.0 `DatabaseMetaData` objects include the following methods to verify this:

- `boolean othersDeletesAreVisible(int) throws SQLException`
- `boolean othersUpdatesAreVisible(int) throws SQLException`
- `boolean othersInsertsAreVisible(int) throws SQLException`

Each takes a result set type, `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_SENSITIVE`, or `ResultSet.TYPE_SCROLL_INSENSITIVE`, as input.

> **Note:** Explicit use of the `refreshRow` method is distinct from this discussion of visibility. For example, even though external updates are invisible to a scroll-insensitive result set, you can explicitly refetch rows in a scroll-insensitive/updatable result set and retrieve external changes that have been made. Visibility refers only to the fact that the scroll-insensitive/updatable result set would not see such changes automatically and implicitly.

Visibility versus Detection of External Changes

Regarding changes made to the 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 change being "visible" 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 change being "detected", 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.

JDBC 2.0 `DatabaseMetaData` objects include the following methods to verify this:

- `boolean deletesAreDetected(int) throws SQLException`
- `boolean updatesAreDetected(int) throws SQLException`
- `boolean insertsAreDetected(int) throws SQLException`

Each takes a result set type, `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_SENSITIVE`, or `ResultSet.TYPE_SCROLL_INSENSITIVE`, as input.

It follows, then, that result set methods specified by JDBC 2.0 to detect changes, `rowDeleted`, `rowUpdated`, and `rowInserted`, will always return `false`. There is no use in calling them.

Summary of Visibility of Internal and External Changes

Table 19–1 summarizes the discussion in the preceding sections regarding whether 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 19–1 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

Notes:

- 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:** Because this kind of refetching is not a highly efficient or optimized methodology, there are 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.

Summary of New Methods for Result Set Enhancements

This section summarizes all the new connection, result set, statement, and database meta data methods added for JDBC 2.0 result set enhancements. This section covers the following methods:

Modified Connection Methods

Following is an alphabetic summary of modified connection methods that allow you to specify result set and concurrency types when you create statement objects:

- `Statement createStatement(int resultSetType, int resultSetConcurrency)`

  This method enables you to specify result set type and concurrency type when you create a generic `Statement` object.

- `CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency)`

  This method enables you to specify result set type and concurrency type when you create a `PreparedStatement` object.

- `PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency)`

  This method enables you to specify result set type and concurrency type when you create a `CallableStatement` object.

New Result Set Methods

Following is an alphabetic summary of new result set methods for JDBC 2.0 result set enhancements:

- `boolean absolute(int row) throws SQLException`

  Move to an absolute row position in the result set.

- `void afterLast() throws SQLException`

  Move to after the last row in the result set.

- `void beforeFirst() throws SQLException`

  Move to before the first row in the result set.

- `void cancelRowUpdates() throws SQLException`

Cancel an `UPDATE` operation on the current row. Call this after the `update`*`XXX`* calls but before the `updateRow` call.

- `void deleteRow() throws SQLException`

  Delete the current row.

- `boolean first() throws SQLException`

  Move to the first row in the result set.

- `int getConcurrency() throws SQLException`

  Returns an `int` value for the concurrency type used for the query.

- `int getFetchSize() throws SQLException`

  Check the fetch size to determine how many rows are fetched in each database round trip.

- `int getRow() throws SQLException`

  Returns the row number of the current row. Returns 0 if there is no valid current row.

- `int getType() throws SQLException`

  Returns an `int` value for the result set type used for the query.

- `void insertRow() throws SQLException`

  Write a result set `INSERT` operation to the database. Call this after calling `update`*`XXX`*`()` methods to set the data values.

- `boolean isAfterLast() throws SQLException`

  Returns `true` if the position is after the last row.

- `boolean isBeforeFirst() throws SQLException`

  Returns `true` if the position is before the first row.

- `boolean isFirst() throws SQLException`

  Returns `true` if the position is at the first row.

- `boolean isLast() throws SQLException`

  Returns `true` if the position is at the last row.

- `boolean last() throws SQLException`

  Move to the last row in the result set.

- `void moveToCurrentRow() throws SQLException`

  Move from the insert-row staging area back to what had been the current row prior to the `moveToInsertRow()` call.

- `void moveToInsertRow() throws SQLException`

  Move to the insert-row staging area to set up a row to be inserted.

- `boolean next() throws SQLException`

  Iterate forward through the result set.

- `boolean previous() throws SQLException`

  Iterate backward through the result set.

- `void refreshRow() throws SQLException`

Refetch the database rows corresponding to the current window in the result set, to update the data. This method is called implicitly for scroll-sensitive result sets.

- `boolean relative(int row) throws SQLException`

  Move to a relative row position, either forward or backward from the current row.

- `void setFetchSize(int rows) throws SQLException`

  Set the fetch size to determine how many rows are fetched in each database round trip when refetching.

- `void updateRow() throws SQLException`

  Write an `UPDATE` operation to the database after using `update`*`XXX`*`()` methods to update the data values.

- `void update`*`XXX`*`() throws SQLException`

  Set or update data values in a row to be updated or inserted. There is an `update`*`XXX`* method for each data type. After calling all the appropriate `update`*`XXX`* methods for the columns to be updated or inserted, call `updateRow` for an `UPDATE` operation or `insertRow` for an `INSERT` operation.

Statement Methods

Following is an alphabetical summary of statement methods for JDBC 2.0 result set enhancements. These methods are available in generic statement, prepared statement, and callable statement objects.

- `int getFetchSize() throws SQLException`

  Check the fetch size to determine how many rows are fetched in each database round trip when executing a query.

- `void setFetchSize(int rows) throws SQLException`

  Set the fetch size to determine how many rows are fetched in each database round trip when executing a query.

- `void setResultSetCache(OracleResultSetCache cache) throws SQLException`

  Use your own client-side cache implementation for scrollable result sets. Create your own class that implements the `OracleResultSetCache` interface, then use the `setResultSetCache` method to input an instance of this class to the statement object that will create the result set.

- `int getResultSetType() throws SQLException`

  Check the result set type of result sets produced by this statement object, which was specified when the statement object was created.

- `int getResultSetConcurrency() throws SQLException`

  Check the concurrency type of result sets produced by this statement object, which was specified when the statement object was created.

Database Meta Data Methods

Following is an alphabetical summary of database meta data methods for JDBC 2.0 result set enhancements.

- `boolean ownDeletesAreVisible(int) throws SQLException`

Returns `true` if, in this JDBC implementation, the specified result set type can see the effect of its own internal `DELETE` operations.

- `boolean ownUpdatesAreVisible(int) throws SQLException`

  Returns `true` if, in this JDBC implementation, the specified result set type can see the effect of its own internal `UPDATE` operations.

- `boolean ownInsertsAreVisible(int) throws SQLException`

  Returns `true` if, in this JDBC implementation, the specified result set type can see the effect of its own internal `INSERT` operations.

- `boolean othersDeletesAreVisible(int) throws SQLException`

  Returns `true` if, in this JDBC implementation, the specified result set type can see the effect of an external `DELETE` operation in the database.

- `boolean othersUpdatesAreVisible(int) throws SQLException`

  Returns `true` if, in this JDBC implementation, the specified result set type can see the effect of an external `UPDATE` operation in the database.

- `boolean othersInsertsAreVisible(int) throws SQLException`

  Returns `true` if, in this JDBC implementation, the specified result set type can see the effect of an external `INSERT` operation in the database.

- `boolean deletesAreDetected(int) throws SQLException`

  Returns `true` if, in this JDBC implementation, the specified result set type can detect when an external `DELETE` operation occurs in the database. This method always returns `false`.

- `boolean updatesAreDetected(int) throws SQLException`

  Returns `true` if, in this JDBC implementation, the specified result set type can detect when an external `UPDATE` operation occurs in the database. This method always returns `false`.

- `boolean insertsAreDetected(int) throws SQLException`

  Returns `true` if, in this JDBC implementation, the specified result set type can detect when an external `INSERT` operation occurs in the database. This method always returns `false`.

20

JDBC RowSets

This chapter contains the following sections:

- Overview
- CachedRowSet
- JDBCRowSet
- WebRowSet
- FilteredRowSet
- JoinRowSet

Overview

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. The support for RowSet was introduced in JDBC 2.0 through the optional packages. However, the implementations of RowSets was standardized in the JDBC RowSet Implementations Specification (JSR-114) by Sun Microsystems, which is available in Java Development Kit (JDK) 5.0. The JDK 5.0 Javadoc provides information about the standard interfaces and base classes for JDBC RowSet implementations.

See Also:

- JSR-114 specification at:
  `http://jcp.org/en/jsr/detail?id=114`
- JDK 5.0 Javadoc at:
  `http://java.sun.com/j2se/1.5.0/docs/api/index.html`

Note: In case of any conflict, the JSR-114 specification takes precedence over the JDK 5.0 Javadoc.

The JSR-114 specification includes implementation details for five types of RowSet:

- CachedRowSet
- JDBCRowSet
- WebRowSet
- FilteredRowSet

- JoinRowSet

Oracle Database 10*g* release 2 (10.2) provides support for implementation of all five types of RowSets through the interfaces and classes packaged in the `javax.sql.rowset` package. In Oracle Database 10*g* release 2 (10.2), this package is included in the standard Oracle JDBC JAR files, `classes12.jar` and `ojdbc14.jar`.

> **Note:** Prior to Oracle Database 10*g* release 2 (10.2), the row set implementation classes were packaged in the `ocrs12.jar` file.

To use the Oracle RowSet implementations, you need to include the standard Oracle JDBC Java Archive (JAR) files in the `CLASSPATH` environment variable and import the `oracle.jdbc.rowset` package or specific classes and interfaces from the package for the required row set type.

> **See Also:** "Checking 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 connect 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://java.sun.com/j2se/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("SCOTT");
rowset.setPassword("TIGER");
rowset.setCommand("SELECT empno, ename, sal FROM emp");
...
```

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 ename, sal FROM emp WHERE empno = ?");
 // setting the employee number input parameter for employee named "KING"
 rowset.setInt(1, 7839);
 rowset.execute();
 ...
```

In the preceding example, the employee number 7839 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 7839.

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 `ResulSet` 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 RowSet feature as they are in the ResultSet feature. In order to make the RowSet 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 RowSet:

```
...
/**
  * 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, "Ashok");
  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 reference implementation of Sun Microsystems. The `OracleCachedRowSet` class in the `ojdbc14.jar` file 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("SCOTT");
rowset.setPassword("TIGER");
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));
}
...
```

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 ("SCOTT");
rowset.setPassword ("TIGER");
rowset.setCommand ("SELECT empno, ename, sal FROM emp");
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 `ojdbc14.jar` implements the standard JSR-114 interface `javax.sql.rowset.JdbcRowSet`.

Table 20–1 shows how the JDBCRowSet interface differs from CachedRowSet interface.

Table 20–1 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("SCOTT");
rowset.setPassword("TIGER");
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 ojdbc14.jar file, 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 from Sun Microsystems, which is at:
`http://java.sun.com/xml/ns/jdbc/webrowset.xsd`

Validation against the schema is enforced, if supported by the specified parser.

See Also:

- The JSR-114 specification at `http://jcp.org/en/jsr/detail?id=114`
- The JDK 5.0 Javadoc at `http://java.sun.com/j2se/1.5.0/docs/api/index.html`

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,"scott","tiger");
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery("select * from emp");

// 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 ojdbc14.jar file 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 an SQLException.

> **Note:** If you are using classes12.jar instead of ojdbc14.jar for FilteredRowSet features, then use OraclePredicate instead of Predicate. The oracle.jdbc.rowset.OraclePredicate interface is Oracle specific and is equivalent to Predicate. This interface is used when the JSR-114 packages are not available.

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, results in an SQLException being thrown.

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 `ojdbc14.jar` file, 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.

> **Note:** If you are using `classes12.jar` instead of `ojdbc14.jar` for JoinRowSet features, then use `OracleJoinable` instead of `Joinable`. The `oracle.jdbc.rowset.OracleJoinable` interface is Oracle specific and is equivalent to `Joinable`. This interface is used when the JSR-114 packages are not available.

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. Currently, only INNER JOIN is supported. The method would throw an SQLException if other types are specified.

```
public int getJoinType() throws SQLException;
```

This method returns an integer value that indicates the JOIN type set on the OracleJoinRowSet object. This method throws an SQLException.

```
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 an SQLException.

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 traverse 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));
...
```

21

Globalization Support

The Oracle Java Database Connectivity (JDBC) drivers provide globalization support, formerly known as National Language Support (NLS). Globalization support enables you 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:

- Providing Globalization Support
- NCHAR, NVARCHAR2, NCLOB and the defaultNChar Property

See Also:

- "Oracle Character Data Types Support" on page 5-10
- *Oracle Database Globalization Support Guide*
- *Oracle Database Reference*

Notes:

- In Oracle Database 10*g*, the `NLS_LANG` variable is no longer part of the JDBC globalization mechanism. Setting it has no effect.
- The JDBC server-side internal driver provides complete globalization support and does not require any globalization extension files.

Providing Globalization Support

The basic Java Archive (JAR) files, `classes12.jar` and `ojdbc14.jar`, contain 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` of your application.

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 - 10.2.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.

NCHAR, NVARCHAR2, NCLOB and the defaultNChar Property

By default, `oracle.jdbc.OraclePreparedStatement` treats all columns as `CHAR`. To insert Java `String` values into `NCHAR`, `NVARCHAR2`, and `NCLOB` columns, applications had to call `setFormOfUse` on each national-language column. However, in Oracle Database 10*g*, if you set the system property `oracle.jdbc.defaultNChar` to true, JDBC treats all character columns as being national-language. The default value for `defaultNChar` is false.

To set `defaultNChar`, you specify the following at the command-line:

```
java -Doracle.jdbc.defaultNChar=true myApplication
```

If you prefer, your application can specify `defaultNChar` as a connection property.

After this property is set, your application can access `NCHAR`, `NVARCHAR2`, or `NCLOB` data without calling `setFormOfUse`. For example:

```
PreparedStatement pstmt =
conn.prepareStatement("insert into TEST values(?,?,?)");
pstmt.setInt(1, 1); // NUMBER column
pstmt.setString(2, myUnicodeString1); // NVARCHAR2 column
pstmt.setString(3, myUnicodeString2); // NCHAR column
pstmt.execute();
```

However, if you set `defaultNChar` to true and then access `CHAR` columns, then the database will implicitly convert all `CHAR` data into `NCHAR`. This conversion has a substantial performance impact. To avoid this, call `setFormOfUse(4,OraclePreparedStatement.FORM_CHAR)` for each `CHAR` column referred to in the statement. For example:

```
PreparedStatement pstmt =
conn.prepareStatement("insert into TEST values(?,?,?)");
pstmt.setInt(1, 1); // NUMBER column
pstmt.setString(2, myUnicodeString1); // NVARCHAR2 column
pstmt.setString(3, myUnicodeString2); // NCHAR column
pstmt.setFormOfUse(4, OraclePreparedStatement.FORM_CHAR);
pstmt.setString(4, myString); // CHAR column
pstmt.execute();
```

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.

Part V

Performance Enhancements

This part consists of chapters that discuss the Oracle Java Database Connectivity (JDBC) features that enhance performance, such as statement caching, implicit connection caching, run-time connection load balancing, and Oracle Call Interface (OCI) connection pooling. It also includes a chapter that provides information on Oracle performance extensions, such as update batching and row prefetching.

Part V contains the following chapters:

- Chapter 22, "Statement Caching"
- Chapter 23, "Implicit Connection Caching"
- Chapter 24, "Run-Time Connection Load Balancing"
- Chapter 25, "Performance Extensions"
- Chapter 26, "OCI Connection Pooling"

22

Statement Caching

This chapter describes the benefits and use of statement caching, an Oracle Java Database Connectivity (JDBC) extension.

This chapter contains the following sections:

- About Statement Caching
- Using Statement Caching

> **Note:** In Oracle9*i* Database 9.2.0 and later releases, Oracle JDBC provides a new statement cache interface and implementation, replacing the application programming interface (API) supported in Oracle9*i* Database release 1 (9.0.1). The previous API is now deprecated.

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. JDBC 3.0 defines a statement-caching interface.

Statement caching can:

- 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:

- Basics of Statement Caching
- Implicit Statement Caching
- Explicit Statement Caching

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 re-initialized and reset to default values, while metadata is saved. Statements are removed from the cache to conform to the maximum size using an 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 22-6

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 22–1 compares the different methods employed in implicit and explicit statement caching.

Table 22–1 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
- Physically Closing a Cached Statement
- Using Implicit Statement Caching
- Using Explicit 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 in effect.

When using the connection cache, you enable statement caching by setting the `MaxStatementsLimit` property on the connection cache. The `MaxStatementsLimit` applies to every connection in the cache. Each connection has its own statement cache with the same maximum size. This provides a global way to throttle the statement cache size for all connections in the cache. Enabling statement caching enables both implicit and explicit statement caching.

Enabling and Disabling Implicit Statement Caching

Enable implicit statement caching in one of two ways:

- Call `setImplicitCachingEnabled(true)` on the connection
- Call `OracleDataSource.getConnection` with the `ImplicitCachingEnabled` property set to `true`. You set `ImplicitCachingEnabled` by calling `OracleDataSource.setImplicitCachingEnabled(true)`

In addition to calling one of these methods, you also need to call `OracleConnection.setStatementCacheSize` on the physical connection. The argument you supply is the maximum number of statements in the cache. An argument of `0` specifies no caching.

Disable implicit statement caching by calling `setImplicitCachingEnabled(false)` on the connection or by setting the `ImplicitCachingEnabled` property to `false`.

To determine whether implicit caching is enabled, call `getImplicitCachingEnabled`, which returns `true` if implicit caching is enabled, `false` otherwise.

Enabling and Disabling Explicit Statement Caching

To enable explicit statement caching you must first set the statement cache size. You set the cache size in one of two ways:

- Call `OracleConnection.setStatementCacheSize` on the physical connection
- Call `OracleDatasource.setMaxStatements`

In either case, 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.

```
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.

Notes:

- 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.

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);
```

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 scheme
- 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 normally.

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 `setDisableStatementCaching` method of the statement object. You can manage cache space by calling the `setDisableStatementCaching` 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 ();
```

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:

```
((OraclePreparedStatement)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 22–2 describes the methods used to allocate statements and retrieve implicitly cached statements.

Table 22–2 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

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 normally.

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 22–3 describes the methods used to retrieve explicitly cached statements.

Table 22–3 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

23

Implicit Connection Caching

Connection caching, generally implemented in the middle tier, is a means of keeping and using cache of physical database connections.

> **Note:** The previous cache architecture, based on `OracleConnectionCache` and `OracleConnectionCacheImpl`, is deprecated. We recommend that you take advantage of the new architecture, which is more powerful and offers better performance.

The implicit connection cache is an improved Java Database Connectivity (JDBC) 3.0-compliant connection cache implementation for `DataSource`. Java and Java2 Platform, Enterprise Edition (J2EE) applications benefit from transparent access to the cache, support for multiple users, and the ability to request connections based on user-defined profiles.

An application turns the implicit connection cache on by calling `setConnectionCachingEnabled(true)` on an `OracleDataSource`. After implicit caching is turned on, the first connection request to the `OracleDataSource` transparently creates a connection cache. There is no need for application developers to write their own cache implementations.

This chapter is divided into the following sections:

- The Implicit Connection Cache
- Using the Connection Cache
- Connection Attributes
- Connection Cache Properties
- Connection Cache Manager API
- Advanced Topics

> **Note:** The concept of connection caching is not relevant to the server-side internal driver, where you always use the default connection. Connection caching is only relevant to the client-side JDBC drivers and the server-side Thin driver.

The Implicit Connection Cache

The connection caching architecture has been redesigned so that caching is transparently integrated into the data source architecture.

The connection cache uses the concept of physical connections and logical connections. Physical connections are the actual connections returned by the database and logical connections are wrappers used by the cache to manipulate physical connections. You can think of logical connections as handles. The caches always return logical connections, which implement the same interfaces as physical connections.

The implicit connection cache offers:

- Driver independence

  Both the JDBC Thin and JDBC Oracle Call Interface (OCI) drivers support the implicit connection cache.

- Transparent access to the JDBC connection cache

  After an application turns implicit caching on, it uses the standard `OracleDataSource` application programming interfaces (APIs) to get connections. With caching enabled, all connection requests are serviced from the connection cache.

  When an application calls `OracleConnection.close` to close the logical connection, the physical connection is returned to the cache.

- Single cache per `OracleDataSource` instance

  When connection caching is turned on, each cache enabled `OracleDataSource` has exactly one cache associated with it. All connections obtained through that data source, no matter what user name and password are used, are returned to the cache. When an application requests a connection from the data source, the cache either returns an existing connection or creates a new connection with matching authentication information.

  Note: Caches cannot be shared between `DataSource` instances. There is a one-to-one mapping between cache enabled `DataSource` instances and caches.

- Heterogeneous user names and passwords per cache

  Unlike in the previous cache implementation, all connections obtained through the same data source are stored in a common cache, no matter what user name and password the connection requests.

- Support for JDBC 3.0 connection caching, including support for multiple users and the required cache properties.

- Property-based configuration

  Cache properties define the behavior of the cache. The supported properties set timeouts, the number of connections to be held in the cache, and so on. Using these properties, applications can reclaim and reuse abandoned connections. The implicit connection cache supports all the JDBC 3.0 connection cache properties.

- OracleConnectionCacheManager

  The new `OracleConnectionCacheManager` class provides a rich set of administrative APIs that applications can use to manage the connection cache. Each virtual machine has one distinguished instance of

`OracleConnectionCacheManager`. Applications manage a cache through the single `OracleConnectionCacheManager` instance.

- User-defined connection attributes

  The implicit connection cache supports user-defined connection attributes that can be used to determine which connections are retrieved from the cache. Connection attributes can be thought of as labels whose semantics are defined by the application, not by the caching mechanism.

- Callback mechanism

  The implicit connection cache provides a mechanism for users to define cache behavior when a connection is returned to the cache, when handling abandoned connections, and when a connection is requested but none is available in the cache.

- Connect-time load balancing

  Implicit connection caching provides connect time load balancing when a connection is first created by the application. The database listener distributes connection creation across Oracle Real Application Cluster instances that would perform the best at the time of connection creation.

  See Also: *Oracle Database Oracle Clusterware and Oracle Real Application Clusters Administration and Deployment Guide*

- Run-time connection load balancing

  Run-time connection load balancing of work requests uses Service Metrics to route work requests to an Oracle Real Application Cluster instance that offers the best performance. Selecting a connection from the cache based on service, to execute a work request, greatly increases the throughput and scalability.

  See Also: Chapter 24, "Run-Time Connection Load Balancing" and *Oracle Database Oracle Clusterware and Oracle Real Application Clusters Administration and Deployment Guide*

Using the Connection Cache

This section discusses how applications use the implicit connection cache. It covers the following topics:

- Turning Caching On
- Opening a Connection
- Setting Connection Cache Name
- Setting Connection Cache Properties
- Closing A Connection
- Implicit Connection Cache Example

Turning Caching On

An application turns the implicit connection cache on by calling `OracleDataSource.setConnectionCachingEnabled(true)`. After implicit caching is turned on, the first connection request to the `OracleDataSource` transparently creates a connection cache.

Example 23–1 provides a sample code that uses the implicit connection cache.

Example 23–1 Using the Implicit Connection Cache

```
// Example to show binding of OracleDataSource to JNDI,
// then using implicit connection cache

import oracle.jdbc.pool.*; // import the pool package

Context ctx = new InitialContext(ht);
OracleDataSource ods = new OracleDataSource();

// Set DataSource properties
ods.setUser("Scott");
ods.setConnectionCachingEnabled(true);    // Turns on caching
ctx.bind("MyDS", ods);
// ...
// Retrieve DataSource from the InitialContext
ods =(OracleDataSource) ctx. lookup("MyDS");

// Transparently create cache and retrieve connection
conn = ods.getConnection();
// ...
conn.close();  // return connection to the cache
// ...
ods.close() // close datasource and clean up the cache
```

Opening a Connection

After you have turned connection caching on, whenever you retrieve a connection through `OracleDataSource.getConnection`, the JDBC drivers check to see if a connection is available in the cache.

The `getConnection` method checks if there are any free physical connections in the cache that match the specified criteria. If a match is found, then a logical connection is returned wrapping the physical connection. If no physical connection match is found, then a new physical connection is created, wrapped in a logical connection, and returned.

There are four variations on `getConnection`, two that make no reference to the connection cache and two that specify which sort of connections the cache may return. The non-cache-specific `getConnection` methods behave as normal.

Note: When implicit connection cache is enabled, the connection returned by `OracleDataSource.getConnection` may not have the state reset. You must, therefore, reset all the connection states, such as auto-commit, batch size, prefetch size, transaction status, and transaction isolation, before putting the connection back into the cache.

Setting Connection Cache Name

The `ConnectionCacheName` property of `OracleDataSource` is an optional property used by the Connection Cache manager to manage a connection cache. You can set this property by calling the following method:

```
public void synchronized setConnectionCacheName(String cacheName) throws
```

`SQLException`

When this property is set, the name is used to uniquely identify the cache accessed by the cache-enabled `OracleDataSource`. If the property is not set, then a default cache name is created using the convention *`DataSourceName#HexRepresentationOfNumberOfCaches`*.

Note: The `getConnectionCacheName()` method will return the name of the connection cache only if the `setConnectionCacheName` method is called after the `setConnectionCachingEnabled` method is called.

Setting Connection Cache Properties

You can fine-tune the behavior of the implicit connection cache using the `setConnectionCacheProperties` method to set various connection properties.

See Also: "Connection Cache Properties" on page 23-8

Note: Although these properties govern the behavior of the connection cache, they are set on the data source, and not on the connection or on the cache itself.

Closing A Connection

An application returns a connection to the cache by calling `close`. There are two variants of the close method: one with no arguments and one that takes a `Connection` object as argument.

Notes:

- Applications must close connections to ensure that the connections are returned to the cache.
- When implicit connection cache is enabled, you must reset all the connection states, such as auto-commit, batch size, prefetch size, transaction status, and transaction isolation, before putting the connection back into the cache. This ensures that any subsequent connection retrieved from the cache will have its state reset.

Implicit Connection Cache Example

Example 23–2 demonstrates creating a data source, setting its caching and data source properties, retrieving a connection, and closing that connection in order to return it to the cache.

Example 23–2 Connection Cache Example

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.naming.*;
import javax.naming.spi.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
```

```
...

    // create a DataSource
    OracleDataSource ods = new OracleDataSource();

    // set cache properties
    java.util.Properties prop = new java.util.Properties();
    prop.setProperty("MinLimit", "2");
    prop.setProperty("MaxLimit", "10");

    // set DataSource properties
    String url = "jdbc:oracle:oci8:@";
    ods.setURL(url);
    ods.setUser("hr");
    ods.setPassword("hr");
    ods.setConnectionCachingEnabled(true); // be sure set to true
    ods.setConnectionCacheProperties (prop);
    ods.setConnectionCacheName("ImplicitCache01"); // this cache's name


    // We need to create a connection to create the cache
    Connection conn = ds.getConnection(user, pass);
    Statement  stmt  = conn1.createStatement();
    ResultSet  rset  = stmt.executeQuery("select user from dual");
    conn1.close();

    ods.close();
```

Connection Attributes

Each connection obtained from a data source can have user-defined attributes. Attributes are specified by the application developer and are `java.lang.Properties` name/value pairs.

An application can use connection attributes to supply additional semantics to identify connections. For instance, an application may create an attribute named `connection_type` and then assign it the value `payroll` or `inventory`.

> **Note:** The semantics of connection attributes are entirely application-defined. The connection cache itself enforces no restrictions on the key or value of connection attributes.

The methods that get and set connection attributes are found on `OracleConnection`. This section covers the following topics:

- Getting Connections
- Setting Connection Attributes
- Checking Attributes of a Returned Connection
- Connection Attribute Example

Getting Connections

The first connection you retrieve has no attributes. You must set them. After you have set attributes on a connection, you can request the connection by specifying its attribute, using the specialized forms of `getConnection`:

- `getConnection(java.util.Properties cachedConnectionAttributes`

  Requests a database connection that matches the specified `cachedConnectionAttributes`.

  > **See Also:** For a discussion on connection attributes, see "Other Properties" on page 23-10.

- `getConnection(java.lang.String user, java.lang.String password, java.util.Properties cachedConnectionAttributes)`

  Requests a database connection from the implicit connection cache that matches the specified `user`, `password` and `cachedConnectionAttributes`. If null values are passed for `user` and `password`, the `DataSource` defaults are used.

Attribute Matching Rules

The rules for matching `connectionAttributes` come in two variations:

- Basic

  In this, the cache is searched to retrieve the connection that matches the attributes. The connection search mechanism as follows:

  1. If an exact match is found, the connection is returned to the caller.
  2. If an exact match is not found and the `ClosestConnectionMatch` data source property is set, then the connection with the closest match is returned. The closest matched connection is one that has the highest number of the original attributes matched. Note that the closest matched connection may match a subset of the original attributes, but does not have any attributes that are not part of the original list. For example, if the original list of attributes is A, B and C, then a closest match may have A and B set, but never a D.
  3. If none of the existing connections match, a new connection is returned. The new connection is created using the user name and password set on the `DataSource`. If `getConnection(String, String, java.util.Properties)` is called, then the user name and password passed as arguments are used to open the new connection.

- Advanced, where attributes may be associated with weights. The connection search mechanism is similar to the basic `connectionAttributes` based search, except that the connections are searched not only based on the `connectionAttributes`, but also using a set of weights that are associated with the keys on the `connectionAttributes`. These weights are assigned to the keys as a one time operation and is supported as a connection cache property, `AttributeWeights`.

Setting Connection Attributes

An application sets connection attributes using:

```
applyConnectionAttributes(java.util.Properties connAttr)
```

No validation is done on `connAttr`. Applying connection attributes is cumulative. Each time you call `applyConnectionAttributes`, the `connAttr` attribute you supply is added to those previously in force.

Checking Attributes of a Returned Connection

When an application requests a connection with specified attributes, it is possible that no match will be found in the connection cache. When this happens, the connection cache creates a connection with no attributes and returns it. The connection cache cannot create a connection with the requested attributes, because the cache manager is ignorant of the semantics of the attributes.

> **Note:** If the `closestConnectionMatch` property has been set, then the cache manager looks for close attribute matches rather than exact matches.

For this reason, applications should always check the attributes of a returned connection. To do this, use the `getUnMatchedConnectionAttributes` method, which returns a list of any attributes that were not matched in retrieving the connection. If the return value of this method is `null`, you know that you must set all the connection attributes.

Connection Attribute Example

Example 23–3 illustrates using connection attributes.

Example 23–3 Using Connection Attributes

```
java.util.Properties connAttr = new java.util.Properties();
connAttr.setProperty("connection_type", "payroll");

// retrieve connection that matches attributes
Connection conn = ds.getConnection(connAttr);
// Check to see which attributes weren't matched
unmatchedProp = ((OracleConnection)conn).getUnMatchedConnectionAttributes();
if ( unmatchedProp != null )
{
// apply attributes to the connection
((OracleConnection)conn).applyConnectionAttributes(connAttr);
}
// verify whether conn contains property after apply attributes
connProp = ((OracleConnection)conn).getConnectionAttributes();
listProperties (connProp);
```

Connection Cache Properties

The connection cache properties govern the characteristics of a connection cache. This section lists the supported connection cache properties. It covers the following topics:

- Limit Properties
- TIMEOUT Properties
- Other Properties
- Connection Property Example

Applications set cache properties in one of the following ways:

- Using the `OracleDataSource` method `setConnectionCacheProperties`
- When creating a cache using `OracleConnectionCacheManager`
- When re-initializing a cache using `OracleConnectionCacheManager`

Limit Properties

These properties control the size of the cache.

InitialLimit

Sets how many connections are created in the cache when it is created or reinitialized. When this property is set to an integer value greater than 0, creating or reinitializing the cache automatically creates the specified number of connections, filling the cache in advance of need.

Default: 0

MaxLimit

Sets the maximum number of connection instances the cache can hold. The default value is `Integer.MAX_VALUE`, meaning that there is no limit enforced by the connection cache, so that the number of connections is limited only by the number of database sessions configured for the database.

Default: `Integer.MAX_VALUE` (no limit)

MaxStatementsLimit

Sets the maximum number of statements that a connection keeps open. When a cache has this property set, reinitializing the cache or closing the data source automatically closes all cursors beyond the specified `MaxStatementsLimit`.

Default: 0

MinLimit

Sets the minimum number of connections the cache maintains.

Default: 0

Notes:

- Setting the `MinLimit` property does not initialize the cache to contain the minimum number of connections. To do this, use the `InitialLimit` property.
- When `InitialLimit` is greater than `MinLimit`, it is possible to have any number of connections specified by `InitialLimit` upto a value specified by `MaxLimit`. Therefore, `InitialLimit` does not depend on `MinLimit`.
- Connections can fall below the minimum limit set on the connection pool when JDBC Fast Connection Failover DOWN events are processed. The processing removes affected connections from the pool. `MinLimit` will be honored as requests to the connection pool ramp up and the number of connections get past the `MinLimit` value.

TIMEOUT Properties

These properties control the lifetime of an element in the cache.

InactivityTimeout

Sets the maximum time a physical connection can remain idle in a connection cache. An idle connection is one that is not active and does not have a logical handle associated with it. When `InactivityTimeout` expires, the underlying physical connection is closed. However, the size of the cache is not allowed to shrink below `minLimit`, if it has been set.

Default: 0 (no timeout in effect)

TimeToLiveTimeout

Sets the maximum time in seconds that a logical connection can remain open. When `TimeToLiveTimeout` expires, the logical connection is unconditionally closed, the relevant statement handles are canceled, and the underlying physical connection is returned to the cache for reuse.

Default: 0 (no timeout in effect)

> **See Also:** "Use Cases for TimeToLiveTimeout and AbandonedConnectionTimeout" on page 23-17

AbandonedConnectionTimeout

Sets the maximum time that a connection can remain unused before the connection is closed and returned to the cache. A connection is considered unused if it has not had SQL database activity.

When `AbandonedConnectionTimeout` is set, JDBC monitors SQL database activity on each logical connection. For example, when `stmt.execute` is called on the connection, a heartbeat is registered to convey that this connection is active. The heartbeats are set at each database execution. If a connection has been inactive for the specified amount of time, the underlying connection is reclaimed and returned to the cache for reuse.

Default: 0 (no timeout in effect)

> **See Also:** "Use Cases for TimeToLiveTimeout and AbandonedConnectionTimeout" on page 23-17

PropertyCheckInterval

Sets the time interval at which the cache manager inspects and enforces all specified cache properties. `PropertyCheckInterval` is set in seconds.

Default: 900 seconds

Other Properties

These properties control miscellaneous cache behaviors.

AttributeWeights

`AttributeWeights` sets the weight for each attribute set on the connection.

> **See Also:** "AttributeWeights" on page 23-16

ClosestConnectionMatch

`ClosestConnectionMatch` causes the connection cache to retrieve the connection with the closest approximation to the specified connection attributes.

See Also: "ClosestConnectionMatch" on page 23-16

ConnectionWaitTimeout

Specifies cache behavior when a connection is requested and there are already `MaxLimit` connections active. If `ConnectionWaitTimeout` is greater than zero, then each connection request waits for the specified number of seconds or until a connection is returned to the cache. If no connection is returned to the cache before the timeout elapses, then the connection request returns `null`.

Default: 0 (no timeout)

LowerThresholdLimit

Sets the lower threshold limit on the cache. The default is 20 percent of the `MaxLimit` on the connection cache. This property is used whenever a `releaseConnection()` cache callback method is registered.

ValidateConnection

Setting `ValidateConnection` to `true` causes the connection cache to test every connection it retrieves against the underlying database. If a valid connection cannot be retrieved, then an exception is thrown.

Default: `false`

Connection Property Example

Example 23–4 demonstrates how an application uses connection properties.

Example 23–4 Using Connection Properties

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.naming.*;
import javax.naming.spi.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
...
  OracleDataSource ds =  (OracleDataSource) ctx.lookup("...");
  java.util.Properties prop = new java.util.Properties ();
  prop.setProperty("MinLimit", "5");     // the cache size is 5 at least
  prop.setProperty("MaxLimit", "25");
  prop.setProperty("InitialLimit", "3"); // create 3 connections at startup
  prop.setProperty("InactivityTimeout", "1800");    //  seconds
  prop.setProperty("AbandonedConnectionTimeout", "900");  //  seconds
  prop.setProperty("MaxStatementsLimit", "10");
  prop.setProperty("PropertyCheckInterval", "60"); // seconds

  ds.setConnectionCacheProperties (prop);  // set properties
  Connection conn = ds.getConnection();
  conn.dosomework();
  java.util.Properties propList=ds.getConnectionCacheProperties();  // retrieve
```

Connection Cache Manager API

`OracleConnectionCacheManager` provides administrative APIs that the middle tier can use to manage available connection caches. The administration methods are:

- createCache
- disableCache
- enableCache
- existsCache
- getCacheNameList
- getCacheProperties
- getNumberOfActiveConnections
- getNumberOfAvailableConnections
- purgeCache
- refreshCache
- reinitializeCache
- removeCache
- setConnectionPoolDataSource

This section also provides an example of using the Connection Cache Manager in Example Of ConnectionCacheManager Use.

createCache

The `createCache` method has the following signatures:

- `void createCache(String cacheName, javax.sql.DataSource ds, java.util.Properties cacheProps)`

  This method creates a new cache identified by a unique cache name. The newly-created cache is bound to the specified `DataSource` object. Cache properties, when specified, are applied to the cache that gets created. When cache creation is successful, the Connection Cache Manager adds the new cache to the list of caches managed. Creating a cache with a user-defined cache name facilitates specifying more meaningful names. For example, dynamic monitoring service (DMS) metrics collected on a per-cache basis could display metrics attached to a meaningful cache name. `createCache` throws an exception, if a cache already exists for the `DataSource` object passed in.

- `String createCache(javax.sql.DataSource ds, java.util.Properties cacheProps)`

  This method creates a new cache using a generated unique cache name and returns this cache name. The standard convention used in cache name generation is *`DataSourceName#HexRepresentationOfNumberOfCaches`*. The semantics are otherwise identical to the previous form.

disableCache

The signature of this method is as follows:

```
void disableCache(String cacheName)
```

This method temporarily disables the cache specified by `cacheName`. This means that, temporarily, connection requests will not be serviced from this cache. However, in-use connections will continue to work uninterrupted.

enableCache

The signature of this method is as follows:

```
void enableCache(String cacheName)
```

This method enables a disabled cache. This is a no-op if the cache is already enabled.

existsCache

The signature of this method is as follows:

```
boolean existsCache(String CacheName)
```

This method checks whether a specific connection cache exists among the list of caches that the Connection Cache Manager handles. Returns `true` if the cache exists, `false` otherwise.

getCacheNameList

The signature of this method is as follows:

```
String[] getCacheNameList()
```

This method returns all the connection cache names that are known to the Connection Cache Manager. The cache names may then be used to manage connection caches using the Connection Cache Manager APIs.

getCacheProperties

The signature of this method is as follows:

```
java.util.properties getCacheProperties(String cacheName)
```

This method retrieves the cache properties for the specified `cacheName`.

getNumberOfActiveConnections

The signature of this method is as follows:

```
int getNumberOfActiveConnections(String cacheName)
```

This method returns the number of checked out connections, connections that are active or busy, and hence not available for use. The value returned is a snapshot of the number of checked out connections in the connection cache at the time the API was processed. It may become invalid quickly.

getNumberOfAvailableConnections

The signature of this method is as follows:

```
int getNumberOfAvailableConnections(String cacheName)
```

This method returns the number of connections in the connection cache, that are available for use. The value returned is a snapshot of the number of connections

available in the connection cache at the time the API was processed. It may become invalid quickly.

isFatalConnectionError

The signature of this method is as follows:

```
boolean isFatalConnectionError(SQLException se)
```

This method is used to differentiate fatal errors from non-fatal errors and may be used in the exception handling modules of the application to trigger connection retries. Some of the errors that are considered fatal are: `ORA-3113`, `ORA-3114`, `ORA-1033`, `ORA-1034`, `ORA-1089`, `ORA-1090`, and `ORA-17002`.

purgeCache

The signature of this method is as follows:

```
void  purgeCache(String cacheName, boolean cleanupCheckedOutConnections)
```

This method removes connections from the connection cache, but does not remove the cache itself. If the `cleanupCheckedOutConnections` parameter is set to `true`, then the checked out connections are cleaned up, as well as the available connections in the cache. If the `cleanupCheckedOutConnections` parameter is set to false, then only the available connections are cleaned up.

refreshCache

The signature of this method is as follows:

```
void refreshCache(String cacheName, int mode)
```

This method refreshes the cache specified by `cacheName`. There are two `modes` supported, `REFRESH_INVALID_CONNECTIONS` and `REFRESH_ALL_CONNECTIONS`. When called with `REFRESH_INVALID_CONNECTIONS`, each `Connection` in the cache is checked for validity. If an invalid `Connection` is found, then the resources of that connection are removed and replaced with a new `Connection`. The test for validity is a simple query as follows:

```
SELECT 1 FROM dual;
```

When called with `REFRESH_ALL_CONNECTIONS`, all available connections in the cache are closed and replaced with new valid physical connections.

reinitializeCache

The signature of this method is as follows:

```
void reinitializeCache(String cacheName, java.util.properties
cacheProperties)
```

This method reinitializes the cache using the specified new set of cache properties. This supports dynamic reconfiguration of caches. The new properties take effect on all newly-created connections, as well as on existing connections that are not in use. When the `reinitializeCache` method is called, all in-use connections are closed. The new cache properties are then applied to all the connections in the cache.

Note: Calling `reinitializeCache` closes all connections obtained through this cache.

removeCache

The signature of this method is as follows:

```
void removeCache(String cacheName, int timeout)
```

This method removes the cache specified by `cacheName`. All its resources are closed and freed. The second parameter is a wait timeout value that is specified in seconds. If the wait timeout value is 0, then all in-use or checked out connections are reclaimed without waiting for the connections in-use to be done. When called with a wait timeout value greater than 0, the operation waits for the specified period of time for checked out connections to be closed before removing the connection cache. This includes connections that are closed based on timeouts specified. Connection cache removal is not reversible.

setConnectionPoolDataSource

The signature of this method is as follows:

```
void setConnectionPoolDataSource(String cacheName, ConnectionPoolDataSource ds)
```

This method enables connections to be created from an external `OracleConnectionPoolDataSource`, instead of the default `DataSource`, for the given connection cache. When such a `ConnectionPoolDataSource` is set, all `DataSource` properties, such as `url`, are derived from this new `DataSource`.

Example Of ConnectionCacheManager Use

Example 23–5 demonstrates the `OracleConnectionCacheManager` interfaces.

Example 23–5 Connection Cache Manager Example

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.naming.*;
import javax.naming.spi.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
...
// Get singleton ConnectionCacheManager instance
  OracleConnectionCacheManager occm =
  OracleConnectionCacheManager.getConnectionCacheManagerInstance();
  String cacheName = "foo";  // Look for a specific cache
  // Use Cache Manager to check # of available connections
  // and active connections
  System.out.println(occm.getNumberOfAvailableConnections(cacheName)
      " connections are available in cache " + cacheName);

  System.out.println(occm.getNumberOfActiveConnections(cacheName)
      + " connections are active");
  // Refresh all connections in cache
  occm.refreshCache(cacheName,
    OracleConnectionCacheManager.REFRESH_ALL_CONNECTIONS);
  // Reinitialize cache, closing all connections
```

```
java.util.Properties newProp = new java.util.Properties();
newProp.setProperty("MaxLimit", "50");
occm.reinitializeCache(cacheName, newProp);
```

Advanced Topics

This section discusses cache functionality that is useful for advanced users, but is not essential to understanding or using the implicit connection cache. This section covers the following topics:

- Attribute Weights And Connection Matching
- Connection Cache Callbacks

Attribute Weights And Connection Matching

There are two connection cache properties that enable the developer to specify which connections in the connection cache are accepted in response to a `getConnection` request. When you set the `ClosestConnectionMatch` property to `true`, you are telling the connection cache manager to return connections that match only some of the attributes you have specified.

If you do not specify `attributeWeights`, then the connection cache manager returns the connection that matches the highest number of attributes. If you specify `attributeWeights`, then you can control the priority the manager uses in matching attributes.

ClosestConnectionMatch

Setting `ClosestConnectionMatch` to `true` causes the connection cache to retrieve the connection with the closest approximation to the specified connection attributes. This can be used in combination with `AttributeWeights` to specify what is considered a closest match.

Default: `false`

AttributeWeights

Sets the weights for each `connectionAttribute`. Used when `ClosestConnectionMatch` is set to `true` to determine which attributes are given highest priority when searching for matches. An attribute with a high weight is given more importance in determining a match than an attribute with a low weight.

`AttributeWeights` contains a set of `Strings` representing key/value pairs. Each key/value pair sets the weights for each `connectionAttribute` for which the user intends to request a connection. Each `String` is in the format written by the `java.util.Properties.Store(OutputStream, String)` method, and thus can be read by the `java.util.Properties.load(InputStream)` method. The `Key` is a `connectionAttribute` and the `Value` is the weight. A weight must be an integer value greater than 0. The default weight is 1.

For example, `TRANSACTION_ISOLATION` could be assigned a weight of 10 and `ROLE` a weight of 5. If `ClosestConnectionMatch` is set to true, when a `connectionAttribute` based connection request is made on the cache, connections with a matching `TRANSACTION_ISOLATION` will be favored over connections with a matching `ROLE`.

Default: No `AttributeWeights`

Connection Cache Callbacks

The implicit connection cache offers a way for the application to specify callbacks to be called by the connection cache. Callback methods are supported with the `OracleConnectionCacheCallback` interface. This callback mechanism is useful to take advantage of the special knowledge of the application about particular connections, supplementing the default behavior when handling abandoned connections or when the cache is empty.

`OracleConnectionCacheCallback` is an interface that must be implemented by the user and registered with `OracleConnection`. The registration API is:

```
public void
registerConnectionCacheCallback(
OracleConnectionCacheCallback cbk, Object usrObj,  int cbkflag);
```

In this interface, `cbk` is the user implementation of the `OracleConnectionCacheCallback` interface. The `usrObj` parameter contains any parameters that the user wants supplied. This user object is passed back, unmodified, when the callback method is called. The `cbkflag` parameter specifies which callback method should be called. It must be one of the following values:

- `OracleConnection.ABANDONED_CONNECTION_CALLBACK`
- `OracleConnection.RELEASE_CONNECTION_CALLBACK`
- `OracleConnection.ALL_CALLBACKS`

When `ALL_CALLBACKS` is set, all the connection cache callback methods are called. For example,

```
// register callback, to invoke all callback methods
((OracleConnection)conn).registerConnectionCacheCallback( new
UserConnectionCacheCallback(),
  new SomeUserObject(),
OracleConnection.ALL_CALLBACKS);
```

An application can register a `ConnectionCacheCallback` on an `OracleConnection`. When a callback is registered, the connection cache calls the `handleAbandonedConnection` method of the callback before reclaiming the connection. If the callback returns true, then the connection is reclaimed. If the callback returns false, then the connection remains active.

The `UserConnectionCacheCallback` interface supports two callback methods to be implemented by the user, `releaseConnection` and `handleAbandonedConnection`.

Use Cases for TimeToLiveTimeout and AbandonedConnectionTimeout

The following are the use cases for the `TimeToLiveTimeout` and `AbandonedConnectionTimeout` timeout mechanisms when used with implicit connection cache. Note that these timeout mechanisms are applicable to the logical connection when it is retrieved from the connection cache.

- The application considers the connections are completely stateless

  When the connections are stateless either of the timeout mechanisms can be used. The connections for which the timeout expires are put back into the connection cache for reuse. These connections are valid for reuse because there is no session state associated with them.

- The application maintains state on each connection, but has a cleanup routine that can render the connections stateless when they are returned to the connection cache

  In this case, `TimeToLiveTimeout` cannot be used. There is no way for the connection cache to ensure that a connection returned to the cache is in a reusable condition.

  However, `AbandonedConnectionTimeout` can be used in this scenario, only if `OracleConnectionCacheCallback` is registered on the connection. The `handleAbandonedConnection` callback method is implemented by the application and ensures that the necessary cleanup is done. The connection is closed when the timeout processing invokes this callback method. The closing of this connection by the call back method causes the connection to be put back into the connection cache in a state where it is reusable.

  See Also: "Connection Cache Callbacks" on page 23-17

- The application maintains state on each connection, but has no control over the connection and, therefore, cannot ensure cleaning up of state for reuse of connections by other applications or users

  The use of either of the timeout mechanism is *not* recommended.

24

Run-Time Connection Load Balancing

Oracle Database 10*g* provides the run-time connection load balancing feature. This chapter contains the following sections:

- Overview
- Run-Time Connection Load Balancing
- Enabling Run-Time Connection Load Balancing

Overview

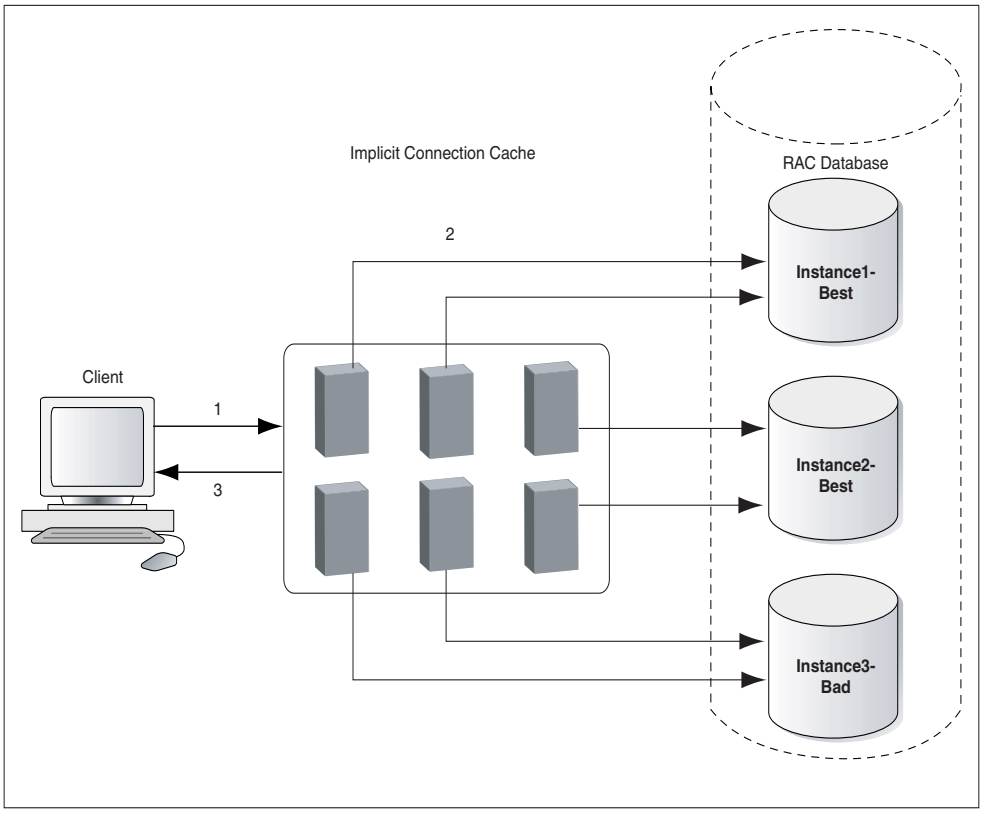
In an Oracle Real Application Clusters environment, a connection could belong to any instance that provides the relevant service. In the best case, all instances perform equally well and randomly retrieving a connection from the cache is appropriate. However, when one instance performs better than others, random selection of a connection is inefficient. 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.

Run-Time Connection Load Balancing

Figure 24–1 illustrates run-time connection load balancing. When run-time connection load balancing is enabled on the implicit connection cache, the following steps occur:

1. A client requests a connection from the connection cache by calling the `getConnection` method on the `DataSource` object.
2. The run-time connection load balancing mechanism selects the connection that belongs to the best instance from the connection cache. In Figure 24–1, this could be either Instance1 or Instance2.
3. The client receives the connection that would process the work request with the best response time.

Figure 24–1 Run-Time Connection Load Balancing



Connection retrieval based on the Load Balancing Advisory is automatic. A request for a connection is serviced by selecting a connection based on the service goal as determined by the Load Balancing Advisory. The service goal determines whether the connection provides best service quality, that is, how efficiently a single transaction completes, or best throughput, that is, how efficiently an entire job or long-running query completes. The advisory is used by the connection cache as long as the events are posted by Oracle Real Application Clusters. When the events stop arriving, the connection cache reverts to random retrieval of connections from the cache.

Run-time connection load balancing relies on the Oracle Notification Service (ONS) infrastructure. It uses the same out-of-band ONS event mechanism that is used for Fast Connection Failover processing. As a result, run-time connection load balancing is enabled by default when Fast Connection Failover is enabled. There is no additional setup or configuration of ONS required to benefit from run-time connection load balancing.

See Also: "Using Fast Connection Failover" on page 27-2

Enabling Run-Time Connection Load Balancing

To enable and use run-time connection load balancing, you must configure the Oracle Real Application Clusters Database in the following manner:

- The service goal must be set to one of the following:
  - `DBMS_SERVICE.SERVICE_TIME`
  - `DBMS_SERVICE.THROUGHPUT`
- The connection balancing goal must be set to `SHORT`.

These goals need to be set when calling `dbms_service.create_service` or `dbms_service.modify_service`. The service goal can be set using the `goal` parameter, and the connection balancing goal can be set using the `clb_goal` parameter.

Note: You can set the connection balancing goal to `LONG`. However, this is mostly uselful for closed workloads, that is, when the rate of completing work is equal to the rate of starting new work.

See Also: *Oracle Database Oracle Clusterware and Oracle Real Application Clusters Administration and Deployment Guide*

25

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

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. The Oracle JDBC driver will throw exceptions when you mix these.

This section covers the following topics:

- Overview of Update Batching Models
- Oracle Update Batching
- Standard Update Batching
- Premature Batch Flush

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.

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.

Oracle update batching is a more efficient model because the driver knows ahead of time how many operations will be batched. In this sense, the Oracle model is more static and predictable. With the standard model, the driver has no way of knowing in advance how many operations will be batched. In this sense, the standard model is more dynamic in nature.

If you want to use update batching, then you can choose between the two models on the basis of the following:

- Use Oracle update batching if portability is not critical. This will probably result in the greatest performance improvement.
- Use standard update batching if portability is a higher priority than performance.

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 each time `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.

Notes:

- 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 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.

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 a stream type, then the Oracle JDBC driver sets the batch value to 1 and sends any queued requests to the database for processing.
- The Oracle JDBC driver automatically runs the `sendBatch` method of an Oracle prepared statement in any of the following circumstances:
  - The connection receives a `COMMIT` request, either as a result of calling the `commit` method or as a result of auto-commit mode.
  - 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);
```

Even though this sets the default batch value for all the prepared statements of the connection, you can override it by calling `setDefaultBatch` on individual Oracle prepared statements.

The connection batch value will apply to statement objects created after this batch value was set.

Note that instead of calling `setDefaultExecuteBatch`, 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 updated 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.

```
ps.setInt (1, 11);
ps.setString (2, "Applications");
ps.setString (3, "Indonesia");

int rows = ps.executeUpdate ();
System.out.println ("Number of rows updated now: " + rows);

ps.close ();
```

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 `commit` 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 `commit` effectively commits changes for all operations that have been added to a batch.

Update Counts in Oracle Batching

In a non-batching situation, the `executeUpdate` method of an `OraclePreparedStatement` object will return 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 will return 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 25–1 illustrates the use of Oracle update batching.

Example 25–1 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("scott");
ods.setPassword("tiger");

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";
```

Standard Update Batching

Oracle implements the standard update batching model according to the JDBC 2.0 specification.

This model, 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 non-batching situation, but whether an operation is added to the batch or the whole batch is processed is typically determined implicitly, depending on whether a pre-determined batch value is reached.

Notes:

- Do not mix standard update batching with Oracle update batching in the same application. The 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

Note the following 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.

- The Oracle implementation of standard update batching does not support stream types as bind values. Any attempt to use stream types will result in an exception.

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.

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();
...
```

The `executeBatch` method returns an `int` array, typically one element per batched operation, indicating success or failure in processing the batch and sometimes containing information about the number of rows affected.

Notes:

- After calling `addBatch`, you must call either `executeBatch` or `clearBatch` before a call to `executeUpdate`, otherwise there will be a SQL exception.
- When a batch is processed, operations are performed in the order in which they were batched.
- The statement batch is reset to empty once `executeBatch` has returned.
- The statement batch is not reset to empty if the connection receives a `ROLLBACK` request. You must explicitly call `clearBatch` to reset it.
- An `executeBatch` call closes the current result set of the statement object, if one exists.

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 non-batched 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.

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();
   ...
}
```

Notes:

- After calling `addBatch`, you must call either `executeBatch` or `clearBatch` before a call to `executeUpdate`, otherwise there will be a SQL exception.
- A `clearBatch` call resets the statement batch to empty.
- Nothing is returned by the `clearBatch` method.

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, it is not possible to know the number of rows affected in the database by each individual statement in the batch. Therefore, all array elements have a value of `-2`. According to the JDBC 2.0 specification, a value of `-2` indicates that the operation was successful but the number of rows affected is unknown.
- 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 Oracle implementation of standard batching.
- For a callable statement batch, the server always returns the value `1` as the update count, irrespective of the number 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 25–2 illustrates the use of standard update batching.

Example 25–2 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, it is not possible to know which operation failed. The array has one element for each operation in the batch, and each element has a value of `-3`. According to the JDBC 2.0 specification, a value of `-3` indicates that an operation did not complete successfully. In this case, it was presumably just one operation that actually failed, but because the JDBC driver does not know which operation that was, it labels all the batched operations as failures.

  You should always perform a `ROLLBACK` operation in this situation.

- 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 Non-Batched Statements

You cannot call `executeUpdate` for regular, non-batched processing of an operation if the statement object has a pending batch of operations.

However, you can intermix batched operations and non-batched operations in a single statement object if you process non-batched 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 legal 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 non-batched 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 regards to update batching operations. A `COMMIT` request will affect all non-batched 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", "SCOTT");
info.setProperty("passwd", "TIGER");
// 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 25–3 illustrates premature batch flushing.

Example 25–3 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 normal 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:

- Oracle Row Prefetching
- Defining Column Types
- DatabaseMetaData TABLE_REMARKS Reporting

Oracle Row Prefetching

Oracle JDBC drivers include extensions that enable you to set the number of rows to prefetch into the client while a result set is being populated during a query. This feature reduces the number of round trips to the server.

This section covers the following topics:

- Setting the Oracle Prefetch Value
- Oracle Row-Prefetching Limitations
- Considerations for getProcedures and getProcedureColumns Methods

> **Note:** With JDBC 2.0, the ability to preset the fetch size became standard functionality.

Setting the Oracle Prefetch Value

Standard JDBC receives the result set one row at a time, and each row requires a round trip to the database. The row-prefetching feature associates an integer row-prefetch setting with a given statement object. JDBC fetches that number of rows at a time from the database during the query. That is, JDBC will fetch *n* rows that match the query criteria and bring them all back to the client at once, where *n* is the prefetch setting. Then, once your `next` calls have run through those *n* rows, JDBC will go back to fetch the next *n* rows that match the criteria.

You can set the number of rows to prefetch for a particular Oracle statement. You can also reset the default number of rows that will be prefetched for all statements in your connection. The default number of rows to prefetch to the client is 10.

Set the number of rows to prefetch for a particular statement as follows:

1. Cast your statement object to `OracleStatement`, `OraclePreparedStatement`, or `OracleCallableStatement`, if necessary.
2. Use the `setRowPrefetch` method of the statement object to specify the number of rows to prefetch, passing in the number as an integer. If you want to check the current prefetch number, then use the `getRowPrefetch` method, which returns an integer.

Set the default number of rows to prefetch for all statements in a connection, as follows:

1. Cast your `Connection` object to `OracleConnection`.

2. Use the `setDefaultRowPrefetch` method of your `OracleConnection` object to set the default number of rows to prefetch, passing in an integer that specifies the desired default. If you want to check the current setting of the default, then use the `getDefaultRowPrefetch` method of the `OracleConnection` object. This method returns an integer.

   Equivalently, instead of calling `setDefaultRowPrefetch`, you can set the `defaultRowPrefetch` Java property if you use a Java `Properties` object in establishing the connection.

   Notes:

   - 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 row-prefetch 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 row-prefetch value will be overridden by any setting of the fetch size.

Example 25–4 illustrates row prefetching.

Example 25–4 Row Prefetching

The following example illustrates the row-prefetching feature. It assumes you have imported the `oracle.jdbc.*` interfaces.

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci:");
ods.setUser("scott");
ods.setPassword("tiger");
Connection conn = ods.getConnection();

//Set the default row-prefetch setting for this connection
((OracleConnection)conn).setDefaultRowPrefetch(7);

/* The following statement gets the default row-prefetch value for
   the connection, that is, 7.
 */
Statement stmt = conn.createStatement();

/* Subsequent statements look the same, regardless of the row
   prefetch value. Only execution time changes.
 */
ResultSet rset = stmt.executeQuery("SELECT ename FROM emp");
System.out.println( rset.next () );

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

//Override the default row-prefetch setting for this statement
( (OracleStatement)stmt ).setRowPrefetch (2);
```

```
ResultSet rset = stmt.executeQuery("SELECT ename FROM emp");
System.out.println( rset.next () );

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

stmt.close();
```

Oracle Row-Prefetching Limitations

There is no maximum prefetch setting, but empirical evidence suggests that 10 is effective. Oracle has never observed a performance benefit to setting prefetch higher than 50. If you do not set the default row-prefetch value for a connection, then 10 is the default.

A statement object receives the default row-prefetch setting from the associated connection at the time the statement object is created. Subsequent changes to the default connection row-prefetch setting have no effect on the statement row-prefetch setting.

If a column of a result set is of data type `LONG` or `LONG RAW`, that is, the streaming types, JDBC changes the statement row-prefetch setting to 1, even if you never actually read a value of either of those types.

You can set the default connection row-prefetch value using a Properties object.

See Also: "Supported Connection Properties" on page 9-7

Defining Column Types

The implementation of `defineColumnType` has changed significantly in 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`. In 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 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);
```

or, if the column is `VARCHAR` or equivalent and you know the length limit:

```
stmt.defineColumnType(column_index, typeCode, max_size);
```

or, 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 );
```

or, 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 25–5 illustrates the use of this feature. It assumes you have imported the `oracle.jdbc.*` interfaces.

Example 25–5 Defining Column Types

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@localhost:1502:orcl");
ods.setUser("scott");
ods.setPassword("tiger");
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 25–1 lists the valid column definition arguments you can use in the `defineColumnType` method.

Table 25–1 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.

DatabaseMetaData TABLE_REMARKS Reporting

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 25–6 illustrates how to enable `TABLE_REMARKS` reporting.

Example 25–6 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.driver.OracleConnection)conn ).setIncludeSynonyms(true)
```

This will cause all subsequent `getColumns` method call 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.

> **See Also:** Table 9–3, " Connection Properties Recognized by Oracle JDBC Drivers"

However, you need to 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 `%`. Otherwise the `catalog` parameter should be a package name pattern, with SQL wild cards, if desired.

- `schemaPattern`

  All objects within Oracle 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 `%`.

26

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:

- OCI Driver Connection Pooling: Background
- OCI Driver Connection Pooling and Shared Servers Compared
- Defining an OCI Connection Pool
- Connecting to an OCI Connection Pool
- Sample Code for OCI Connection Pooling
- Statement Handling and Caching
- JNDI and the OCI Connection Pool

> **Note:** Use OCI connection pooling if you need session multiplexing. Otherwise, Oracle recommends using the Implicit Connection Cache functionality.

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 normally 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 behave as an 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 case of shared servers, the connection from the client is normally 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
    ("SCOTT", "TIGER", "jdbc:oracle:oci:@(description=(address=(host=
    myhost)(protocol=tcp)(port=1521))(connect_data=(INSTANCE_NAME=orcl)))",
    poolConfig);
```

`poolConfig` is a set of properties which 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("SCOTT");
cpool.setPassword("TIGER");
cpool.setURL("jdbc:oracle:oci:@(description=(address=(host=
    myhost)(protocol=tcp)(port=1521))(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) scheme.

- `String getNoWait()`

  Retrieves whether 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 whether 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)`

  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 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
    {
      DriverManager.registerDriver(new OracleDriver());

      String s = null;   //System.getProperty ("JDBC_URL");

      //String url = ( s == null ? "jdbc:oracle:oci8:@orcl" : s);
      String url = "jdbc:oracle:oci8:@orcl.rmmslang.com";

      OracleOCIConnectionPool cpool = new OracleOCIConnectionPool("scott",
"tiger", 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("scott", "tiger");
        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 we 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 persistent the properties of Java object, 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 persistent the properties, 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.

Part VI

High Availability

This section provides information about the high-availability features of Oracle Database 10*g*. It discusses the Fast Connection Failover and Transparent Application Failover (TAF) features

Part VI contains the following chapters:

- Chapter 27, "Fast Connection Failover"
- Chapter 28, "Transparent Application Failover"

27

Fast Connection Failover

The Fast Connection Failover mechanism depends on the implicit connection cache feature. As a result, for Fast Connection Failover to be available, implicit connection caching must be enabled.

This chapter is divided into the following sections:

- Introduction
- Using Fast Connection Failover
- Understanding Fast Connection Failover
- Comparison of Fast Connection Failover and TAF

See Also: Chapter 23, "Implicit Connection Caching"

Introduction

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 10*g*. The advantages of Fast Connection Failover include:

- Driver independence

  Fast Connection Failover supports both the JDBC Thin and JDBC Oracle Call Interface (OCI) drivers.

- Integration with implicit connection cache

  The two features work together synergistically to improve application performance and high availability.

- Integration with Oracle Real Application Clusters (RAC)

  This provides superior Real Application Clusters/High Availability event notification mechanisms.

- Easy integration with application code

  You only need to enable Fast Connection Failover and no further configuration is required.

Fast Connection Failover Features

When enabled, Fast Connection Failover provides:

- Rapid detection and cleanup of invalid cached connections, that is, DOWN event processing
- Load balancing of available connections, that is, UP event processing

- Run-time work request distribution to all active RAC instances

Using Fast Connection Failover

Applications manage Fast Connection Failover through `DataSource` instances.

This section covers the following topics:

- Fast Connection Failover Prerequisites
- Configuring ONS For Fast Connection Failover
- Enabling Fast Connection Failover
- Querying Fast Connection Failover Status

Fast Connection Failover Prerequisites

Fast Connection Failover is available under the following circumstances:

- The implicit connection cache is enabled

  Fast Connection Failover works in conjunction with the JDBC connection caching mechanism. This helps applications manage connections to ensure high availability.

- The application uses service names to connect to the database

  The application cannot use service identifiers (SIDs).

- The underlying database has Oracle Database 10*g* Real Application Clusters capability

  If failover events are not propagated, then connection failover cannot occur.

- Oracle Notification Service (ONS) is configured and available on the node where JDBC is running

  JDBC depends on ONS to propagate database events and notify JDBC of them.

- The Java virtual machine (JVM) in which your JDBC instance is running must have `oracle.ons.oraclehome` set to point to your *`ORACLE_HOME`*.

Configuring ONS For Fast Connection Failover

In order for Fast Connection Failover to work, you must configure ONS correctly. ONS is shipped as part of Oracle Database 10*g*.

This section covers the following topics:

- ONS Configuration File
- Client-Side ONS Configuration
- Server-Side ONS Configuration Using racgons
- Remote ONS Subscription

ONS Configuration File

ONS configuration is controlled by the ONS configuration file, *`ORACLE_HOME`*`/opmn/conf/ons.config`. This file tells the ONS daemon details about how it should behave and who it should talk to. Configuration information within `ons.config` is defined in simple name/value pairs. There are three values that should always be configured within `ons.config`. The first is `localport`, the

port that ONS binds to on the localhost interface to talk to local clients. An example of the `localport` configuration is:

```
localport=4100
```

The second value is `remoteport`, the port that ONS binds to on all interfaces for talking to other ONS daemons. An example of the `remoteport` configuration is:

```
remoteport=4200
```

The third value specifies `nodes`, a list of other ONS daemons to talk to. Node values are given as a comma-delimited list of either hostnames or IP addresses plus ports. Note that the port value that is given is the remote port that each ONS instance is listening on. In order to maintain an identical file on all nodes, the `host:port` of the current ONS node can also be listed in the nodes list. It will be ignored when reading the list.

The nodes listed in the nodes line correspond to the individual nodes in the RAC instance. Listing the nodes ensures that the middle-tier node can communicate with the RAC nodes. At least one mid-tier node and one node in the RAC instance must be configured to see one another. As long as one node on each side is aware of the other, all nodes are visible. You need not list every single cluster and middle-tier node in the ONS config file of each RAC node. In particular, if one ONS config file cluster node is aware of the middle tier, then all nodes in the cluster are aware of it.

An example of the nodes configuration is:

```
nodes=myhost.example.com:4200,123.123.123.123:4200
```

There are also several optional values that can be provided in `ons.config`.

The first optional value is a `loglevel`. This specifies the level of messages that should be logged by ONS. This value is an integer that ranges from `1`, which indicates least messages logged, to `9`, which indicates most messages logged. The default value is `3`. An example is:

```
loglevel=3
```

The second optional value is a `logfile` name. This specifies a log file that ONS should use for logging messages. The default value for `logfile` is *`$ORACLE_HOME`*`/opmn/logs/ons.log`. An example is:

```
logfile=/private/oraclehome/opmn/logs/myons.log
```

The third optional value is a `walletfile` name. A wallet file is used by the Oracle Secure Sockets Layer (SSL) to store SSL certificates. If a wallet file is specified to ONS, it will use SSL when communicating with other ONS instances and require SSL certificate authentication from all ONS instances that try to connect to it. This means that if you want to turn on SSL for one ONS instance, then you must turn it on for all instances that are connected. This value should point to the directory where your `ewallet.p12` file is located. An example is:

```
walletfile=/private/oraclehome/opmn/conf/ssl.wlt/default
```

One optional value is reserved for use on the server side. `useocr=on` is used to tell ONS to store all RAC nodes and port numbers in Oracle Cluster Registry (OCR) instead of in the ONS configuration file. Do not use this option on the client side.

The `ons.config` file allows blank lines and comments on lines that begin with `#`.

Client-Side ONS Configuration

You can access the client-side ONS through *ORACLE_HOME*/opmn. On the client side, there are two ways to set up ONS:

- Remote ONS configuration

  See Also: "Remote ONS Subscription" on page 27-5

- ONS deamon on the client side

Example 27–1 illustrates how a sample configuration file may look like.

Example 27–1 ons.config file

```
# This is an example ons.config file
#
# The first three values are required
localport=4100
remoteport=4200
nodes=racnode1.example.com:4200,racnode2.example.com:4200
```

After configuring ONS, you start the ONS daemon with the `onsctl` command. It is the user's responsibility to make sure that an ONS daemon is running at all times.

Using the onsctl Command

After configuring, use *ORACLE_HOME*/opmn/bin/onsctl to start, stop, reconfigure, and monitor the ONS daemon. Table 27–1 is a summary of the commands that `onsctl` supports.

Table 27–1 onsctl commands

Command	Effect	Output
`start`	Starts the ONS daemon	`onsctl: ons started`
`stop`	Stops the ONS daemon	`onsctl: shutting down ons daemon...`
`ping`	Verifies whether the ONS daemon is running	`ons is running ...`
`reconfig`	Triggers a reload of the ONS configuration without shutting down the ONS daemon	
`help`	Prints a help summary message for onsctl	
`detailed`	Prints a detailed help message for onsctl	

Server-Side ONS Configuration Using racgons

You can access the server-side ONS through *ORA_CRS_HOME*/opmn. You configure the server side by using `racgons` to add the middle-tier node information to OCR. This command is found in `ORA_CRS_HOME/bin/racgons`. Before using `racgons`, you must edit `ons.config` to set `useocr=on`.

The middle-tier nodes should be configured in OCR, so that all nodes share the configuration, and no matter which RAC nodes are up they can communicate to the mid-tier. When running on a cluster, always configure the ONS hosts and ports not by using the ONS configuration files but using `racgons`. The `racgons` command stores the ONS hosts and ports in OCR, where every node can see it. That way, you don't

need to edit a file on every node to change the configuration, just run a single command on one of the cluster nodes.

The `racogns` command enables you to specify hosts and ports on one node, then propagate your changes among all nodes in a cluster. The command takes two forms:

```
racgons add_config hostname:port [hostname:port] [hostname:port] ...
racgons remove_config hostname[:port] [hostname:port] [hostname:port] ...
```

The `add_config` version adds the listed hostname(s), the `remove_config` version removes them. Both commands propagate the changes among all instances in a cluster.

If multiple port numbers are configured for a host, the specified port number is removed from *`hostname`*. If only *`hostname`* is specified, all port numbers for that host are removed.

> **See Also:** *Oracle Database Oracle Clusterware and Oracle Real Application Clusters Administration and Deployment Guide*

Other Uses of racgons

You should run `racgons` whenever you add a new node to the cluster.

Remote ONS Subscription

The advantages of remote ONS subscription are:

- Support for an All Java mid-tier stack
- No ONS daemon needed on the client computer and, therefore, no need to manage this process
- Simple configuration using the `DataSource` property.

When using remote ONS subscription for Fast Connection Failover, the application invokes the following method on an `OracleDataSource` instance:

```
setONSConfiguration(String remoteONSConfig)
```

The `remoteONSConfig` parameter is a list of name/value pairs of the form *`name=value`* that are separated by a new line character (\n). *`name`* can be one of `nodes`, `walletfile`, or `walletpassword`. This parameter should specify at least the `nodes` ONS configuration attribute, which is a list of *`host:port`* pairs separated by comma (,). The hosts and ports denote the remote ONS daemons available on the RAC nodes.

> **See Also:** "ONS Configuration File" on page 27-2

SSL could be used in communicating with the ONS daemons when the `walletfile` attribute is specified as an Oracle wallet file. In such cases, if the `walletpassword` attribute is not specified, single sign-on (SSO) would be assumed.

Following are a few examples, assuming `ods` is an `OracleDataSource` instance:

```
ods.setONSConfiguration("nodes=racnode1.example.com:4200,racnode2.example.com:4200
");

ods.setONSConfiguration("nodes=racnode1:4200,racnode2:4200\nwalletfile=/mydir/Wall
et\nwalletpassword=mypasswd");

ods.setONSConfiguration("nodes=racnode1:4200,racnode2:4200\nwalletfile=/mydir/conf
/Wallet");
```

> **Note:** The `ons.jar` must be in the `CLASSPATH` on the client. In the case of Oracle Application Server, ONS is embedded in OPMN, as before, and JDBC Fast Connection Failover continues to work as before.

Enabling Fast Connection Failover

An application enables Fast Connection Failover by calling `setFastConnectionFailoverEnabled(true)` on a `DataSource` instance, before retrieving any connections from that instance.

You cannot enable Fast Connection Failover when reinitializing a connection cache. You must enable it before using the `OracleDataSource` instance.

Example 27–2 illustrates how to enable Fast Connection Failover.

> **Note:** After a cache is Fast Connection Failover-enabled, you cannot disable Fast Connection Failover during the lifetime of that cache.

To enable Fast Connection Failover, you must:

- Configure and start ONS. If ONS is not correctly set up, then implicit connection cache creation fails and an `ONSException` is thrown at the first `getConnection` request.
- Set the `FastConnectionFailoverEnabled` property before making the first `getConnection` request to an `OracleDataSource`. When Fast Connection Failover is enabled, the failover applies to all connections in the connection cache. If your application explicitly creates a connection cache using the Connection Cache Manager, then you must first set `FastConnectionFailoverEnabled` before retrieving any connections.
- Use a service name rather than an SID when setting the `OracleDataSource` `url` property.

Example 27–2 Enabling Fast Connection Failover

```
// declare datasource
ods.setUrl(
"jdbc:oracle:oci:@(DESCRIPTION=
  (ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias)
    (PORT=1521))
    (CONNECT_DATA=(SERVICE_NAME=service_name)))");
ods.setUser("scott");
ods.setConnectionCachingEnabled(true);
ods.setFastConnectionFailoverEnabled(true):
ctx.bind("myDS",ods);
ds=(OracleDataSource) ctx.lookup("MyDS");
try {
 ds.getConnection();  // transparently creates and accesses cache
 catch (SQLException SE {
  }
}
...
```

Querying Fast Connection Failover Status

An application determines whether Fast Connection Failover is enabled by calling `OracleDataSource.getFastConnectionFailoverEnabled`, which returns `true` if failover is enabled, `false` otherwise.

Understanding Fast Connection Failover

After Fast Connection Failover is enabled, the mechanism is automatic; no application intervention is needed. This section discusses how a connection failover is presented to an application and what steps the application takes to recover.

This section covers the following topics:

- What The Application Sees
- How It Works

What The Application Sees

When a RAC service failure is propagated to the JDBC application, the database has already rolled back the local transaction. The cache manager then cleans up all invalid connections. When an application holding an invalid connection tries to do work through that connection, it receives `SQLException`, `ORA-17008, Closed Connection`.

When an application receives a `Closed Connection` error message, it should:

1. Retry the connection request. This is essential, because the old connection is no longer open.
2. Replay the transaction. All work done before the connection was closed has been lost.

> **Note:** The application should not try to roll back the transaction. The transaction was already rolled back in the database by the time the application received the exception.

How It Works

Under Fast Connection Failover, each connection in the cache maintains a mapping to a service, instance, database, and hostname.

When a database generates a RAC event, that event is forwarded to the JVM in which JDBC is running. A daemon thread inside the JVM receives the RAC event and passes it on to the Connection Cache Manager. The Connection Cache Manager then throws SQL exceptions to the applications affected by the RAC event.

A typical failover scenario may work like this:

1. A database instance fails, leaving several stale connections in the cache.
2. The RAC mechanism in the database generates a RAC event which is sent to the JVM containing JDBC.
3. The daemon thread inside the JVM finds all the connections affected by the RAC event, notifies them of the closed connection through SQL exceptions, and rolls back any open transactions.
4. Each individual connection receives a SQL exception and must retry.

Comparison of Fast Connection Failover and TAF

Fast Connection Failover differs from Transparent Application Failover (TAF) in the following ways:

- Application-level connection retries

  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. TAF supports connection retries only at the OCI/Net layer.

- Integration with the implicit connection cache

  Fast Connection Failover is well-integrated with the implicit connection cache, which allows the connection cache manager to manage the cache for high availability. For example, failed connections are automatically invalidated in the cache. TAF works at the network level on a per-connection basis, which means that the connection cache cannot be notified of failures.

- Event-based

  Fast Connection Failover is based on the 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 RAC instances.

Note: Oracle recommends *not* to use TAF and Fast Connection Failover in the same application.

28

Transparent Application Failover

This chapter contains the following sections:

- Overview
- Failover Type Events
- TAF Callbacks
- Java TAF Callback Interface

Overview

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 shutdown, 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.

Notes:

- TAF is always active and does not have to be set.
- TAF does not work with the OCI Connection Pool.

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.

Part VII

Transaction Management

This part provides information on 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 chapters:

- Chapter 29, "Distributed Transactions"

29

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:

- Overview
- XA Components
- Error Handling and Optimizations
- Implementing a Distributed Transaction
- Native-XA in Oracle JDBC Drivers

> **Note:** This chapter discusses features of the JDBC 2.0 Optional Package, formerly known as the JDBC 2.0 Standard Extension application programming interface (API), which is available through the `javax` packages from Sun Microsystems. The Optional Package is part of Java Development Kit (JDK) 1.4. For JDK 1.2.x and 1.3.x, the relevant packages are included in the `classes12.jar` file.

For further introductory and general information about distributed transactions, refer to the Sun Microsystems specifications for the JDBC 2.0 Optional Package and the Java Transaction API (JTA).

Overview

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 the JDBC 2.0 extension API, 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
- Distributed Transaction Concepts
- Switching Between Global and Local Transactions
- Oracle XA Packages

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`. This file is located at *`ORACLE_HOME`*`/jlib`. Oracle includes this file with the JDBC product. You can also obtain it from the Sun Microsystems Web site, but it is advisable to use the version from Oracle, because that has been tested with the Oracle drivers.

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.

  Because each `OracleXAResource` instance corresponds to a single Oracle session, 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

As of JDBC 3.0, 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 29–1 describes the connection mode transition rules.

Table 29–1 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.	`start` is called on an `XAResource` obtained from the `XAconnection` that provided this 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	NEVER

Table 29–1 (Cont.) Connection Mode Transitions

Current Mode	Switches To NO_TXN When	Switches to LOCAL_TXN When	Switches To GLOBAL_TXN When
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 are 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 start, 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 an 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, standard XA interfaces specified in the JDBC 2.0 Optional Package, and the Oracle classes that implement them. The following topics are covered:

- XADatasource Interface and Oracle Implementation
- XAConnection Interface and Oracle Implementation
- XAResource Interface and Oracle Implementation
- OracleXAResource Method Functionality and Input Parameters
- Xid Interface and Oracle Implementation

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 non-pooling data sources.

Fast Connection Failover can be enabled on an `OracleXADataSource` object by calling `setFastConnectionFailoverEnabled(true)`. When enabled on an `OracleXADataSource` object, Fast Connection Failover works no differently from when it is enabled on an `OracleDataSource` object.

> **See Also:** For information on Fast Connection Failover, see Chapter 27, "Fast Connection Failover".

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*. In 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.

Notes:

- 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.

The Oracle JDBC driver creates and returns an `OracleXAResource` instance whenever the `getXAResource` method of the `OracleXAConnection` class is called, and it is the 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

Start 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`

  Flag 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

  Join subsequent operations in the session associated with this XA resource instance to the existing transaction branch specified by xid.

- XAResource.TMRESUME

  Resume the transaction branch specified by xid.

  Note: A transaction branch can be resumed only if it had been suspended earlier.

- OracleXAResource.ORATMSERIALIZABLE

  Start a serializable transaction with transaction ID xid.

- OracleXAResource.ORATMREADONLY

  Start a read-only transaction with transaction ID xid.

- OracleXAResource.ORATMREADWRITE

  Start a read/write transaction with transaction ID xid.

- OracleXAResource.ORATRANSLOOSE

  Start a loosely-coupled transaction with transaction ID xid.

TMNOFLAGS, TMJOIN, TMRESUME, 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.

Notes:

- 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 which distributed transaction the transaction branch should belong to. The mechanics of this are handled between the middle tier and transaction manager.

End

End 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.

Notes:

- 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

Prepare 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 encounter 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.

Notes:

- 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

Commit 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.

Roll back

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`.

Check for same RM

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
```

Where `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. Oracle subclasses `XAException` with the `oracle.jdbc.xa.OracleXAException` class.

An `OracleXAException` instance consists of an Oracle error portion and an XA error portion and 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 29–2.

Table 29–2 Oracle-XA Error Mapping

Oracle Error Code	XA Error Code
ORA 3113	XAException.XAER_RMFAIL
ORA 3114	XAException.XAER_RMFAIL
ORA 24756	XAException.XAER_NOTA
ORA 24764	XAException.XA_HEURCOM
ORA 24765	XAException.XA_HEURRB

Table 29–2 (Cont.) Oracle-XA Error Mapping

Oracle Error Code	XA Error Code
ORA 24766	XAException.XA_HEURMIX
ORA 24767	XAException.XA_RDONLY
ORA 25351	XAException.XA_RETRY
all other ORA errors	XAException.XAER_RMERR

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 are subclassed by XA-related classes.

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=dlsun991)
                     (protocol=tcp)(port=5521))(connect_data=(sid=rdbms2)))";
        // Create first DataSource and get connection
        OracleDataSource ods1 = new OracleDataSource();
        ods1.setURL(URL1);
        ods1.setUser("scott");
        ods1.setPassword("tiger");
        Connection conna = ods1.getConnection();

        // Create second DataSource and get connection
        OracleDataSource ods2 = new OracleDataSource();
        ods2.setURL(URL2);
        ods2.setUser("scott");
        ods2.setPassword("tiger");
        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("scott");
oxds1.setPassword("tiger");

OracleXADataSource oxds2 = new OracleXADataSource();

oxds2.setURL("jdbc:oracle:thin:@(description=(address=(host=dlsun991)
           (protocol=tcp)(port=5521))(connect_data=(sid=rdbms2)))");
oxds2.setUser("scott");
oxds2.setPassword("tiger");

// 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 PL/SQL procedures. However, this method is very slow.
- Through native APIs. This method is faster than using PL/SQL procedures.

There is a huge performance difference between the two methods of sending XA commands to the server. The use of native APIs provides high performance gains as compared to the use of PL/SQL procedures.

Prior to Oracle Database 10*g*, the Thin driver used PL/SQL procedures to send XA commands to the server because Thin native APIs were not available. In Oracle Database 10*g*, the Thin native APIs are available and are used by default to send XA commands.

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 9–2, " Oracle Extended Data Source Properties" on page 9-4 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 Sun Solaris or Linux system, you need the shared libraries, `libheteroxa10.so` and `libheteroxa10_g.so`, to enable the Native XA feature. In order for the Native XA feature to work properly, these libraries need to be installed and available in the Sun Solaris search path.

On a Microsoft Windows system, you need the `heteroxa10.dll` and `heteroxa10_g.dll` files to enable the Native XA feature. These files need to be installed and available in the Windows DLL path for the Native XA feature to work properly.

Note: Libraries with the `_g` suffix are debug libraries.

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 C-9 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 end-to-end metrics support in Oracle Java Database Connectivity (JDBC) drivers.

Part VIII contains the following chapters:

- Chapter 30, "End-To-End Metrics Support"

30

End-To-End Metrics Support

Oracle Java Database Connectivity (JDBC) now supports end-to-end metrics when used with Oracle Database 10*g*. This chapter discusses end-to-end metric support. It contains the following sections:

- Introduction
- JDBC API For End-To-End Metrics

Introduction

JDBC supports four end-to-end metrics, all of which are set on a per-connection basis:

- Action

  This is a `String`
- ClientId

  This is a `String`
- ExecutionContextId

  This is a combination of `String` and `short SequenceNumber`)
- Module

  This is a `String`

All of these metrics are set on a per-connection basis. All operations on a given connection share the same values. Applications normally set these metrics using Dynamic Monitoring Service (DMS). Although it is also possible to set metrics using JDBC, metrics set using DMS override metrics set using JDBC. To use DMS directly, you must be using a DMS-enabled Java Archive (JAR), which is only available as part of Oracle Application Server.

Table 30–1 lists the maximum size for each end-to-end metric.

> **Note:** If you are using a DMS-enabled JDBC JAR file, then you must include the JAR file for DMS, `dms.jar`, in your `CLASSPATH`. The DMS-enabled JDBC JAR file and the DMS JAR file must come from the same Oracle Database release.

Table 30–1 Maximum Lengths for End-to-End Metrics

Metric	Maximum length
ACTION	32

Table 30–1 (Cont.) Maximum Lengths for End-to-End Metrics

Metric	Maximum length
CLIENTID	64
ECID (string component)	64
MODULE	48

When a connection is created, the JDBC drivers check DMS for end-to-end metrics. It only makes this check once during the lifetime of the connection.

If DMS metrics are not set, then JDBC never checks DMS for metrics again. Thereafter, each time JDBC communicates with the database, it sends any updated metric values to the database.

If DMS metrics are set, then JDBC ignores the end-to-end metric application programming interface (API) described in this chapter. Thereafter, each time JDBC communicates with the database, it checks with DMS for updated metric values, and, if it finds them, propagates them to the database.

If no metrics are set, then no metrics are sent to the database.

JDBC API For End-To-End Metrics

If DMS is not in use, either because a non-DMS JAR is in use or because no metric values were set in DMS, then the JDBC API is used.

The JDBC API defines the following constants and methods on `OracleConnection`:

- `String[] getEndToEndMetrics() throws SQLException`

  Returns the end-to-end metrics.

- `void setEndToEndMetrics(String[] metrics, short sequenceNumber) throws SQLException`

  Sets the end-to-end metrics.

- `END_TO_END_ACTION_INDEX`

  The index of the `ACTION` metric within the `String` array of metrics.

- `END_TO_END_CLIENTID_INDEX`

  The index of the `CLIENTID` metric within the `String` array of metrics.

- `END_TO_END_MODULE_INDEX`

  The index of the `MODULE` metric within the `String` array of metrics.

- `END_TO_END_ECID_INDEX`

  The index of the string component of the execution context (`ECID`) metric within the `String` array of metrics. This component is not used by Oracle Database 10*g*.

- `END_TO_END_STATE_INDEX_MAX`

  This is the size of the `String` array containing the metric values.

- `short getEndToEndECIDSequenceNumber()`

  Returns the current value of the `SequenceNumber` component of the `ECID`. This component is not used by Oracle Database 10*g*.

To unset the metrics, pass an array of appropriate size with all null values and the value `Short.MIN_VALUE` as the sequence number.

Example 30–1 illustrates how to use JDBC API for end-to-end metrics.

Example 30–1 Using the JDBC API for End-to-End Metrics

```
ods.setUrl(
"jdbc:oracle:oci:@(DESCRIPTION=
  (ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias)
    (PORT=1521))
    (CONNECT_DATA=(SERVICE_NAME=service_name)))");
ods.setUser("scott");
Connection conn = ods.getConnection();

String metrics[] = new String[OracleConnection.END_TO_END_STATE_INDEX_MAX];
metrics[END_TO_END_ACTION_INDEX] = "Spike";
metrics[END_TO_END_MODULE_INDEX] = "Buffy";
// Set these metrics
conn.setEndToEndMetrics(metrics, (short) 0);
// Do some work
// Update a metric
metrics[END_TO_END_MODULE_INDEX] = "Faith";

conn.setEndToEndMetrics(metrics, (short) 0);
// Retrieve metrics
new String[] newMetrics = conn.getEndToEndMetrics();
```

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:

- Appendix A, "Reference Information"
- Appendix B, "Coding Tips"
- Appendix C, "JDBC Error Messages"
- Appendix D, "Troubleshooting"

A

Reference Information

This appendix contains detailed Java Database Connectivity (JDBC) reference information, including the following topics:

- Valid SQL-JDBC Data Type Mappings
- Supported SQL and PL/SQL Data Types
- Embedded SQL92 Syntax
- Oracle JDBC Notes and Limitations

Valid SQL-JDBC Data Type Mappings

Table 13–1 describes the default mappings between Java classes and SQL data types supported by the Oracle JDBC drivers. Compare the contents of the JDBC Type Codes, Standard Java Types, and SQL Data Types columns in Table 13–1 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. The 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.

Notes: For classes where `oracle.sql.ORAData` appears in italic, these can be generated by JPublisher.

Table A–1 Valid SQL Data Type-Java Class Mappings

These SQL data types:	Can be materialized as these Java types:
`CHAR, VARCHAR2, LONG`	`oracle.sql.CHAR`
	`java.lang.String`
	`java.sql.Date`
	`java.sql.Time`
	`java.sql.Timestamp`
	`java.lang.Byte`
	`java.lang.Short`
	`java.lang.Integer`
	`java.lang.Long`

Table A–1 (Cont.) Valid SQL Data Type-Java Class Mappings

These SQL data types:	Can be materialized as these Java types:
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
	byte, short, int, long, float, double
DATE	oracle.sql.DATE
	java.sql.Date
	java.sql.Time
	java.sql.Timestamp
	java.lang.String
NUMBER	oracle.sql.NUMBER
	java.lang.Byte
	java.lang.Short
	java.lang.Integer
	java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
	byte, short, int, long, float, double
OPAQUE	oracle.sql.OPAQUE
RAW, LONG RAW	oracle.sql.RAW
	byte[]
ROWID	oracle.sql.CHAR
	oracle.sql.ROWID
	java.lang.String
BFILE	oracle.sql.BFILE
BLOB	oracle.sql.BLOB
	java.sql.Blob
CLOB	oracle.sql.CLOB
	java.sql.Clob
TIMESTAMP	java.sql.Date, oracle.sql.DATE, java.sql.Time, java.sql.Timestamp, oracle.sql.TIMESTAMP, java.lang.String, byte[]
TIMESTAMP WITH TIME ZONE	java.sql.Date, oracle.sql.DATE, java.sql.Time, java.sql.Timestamp, oracle.sql.TIMESTAMPTZ, java.lang.String, byte[]
TIMESTAMP WITH LOCAL TIME ZONE	java.sql.Date, oracle.sql.DATE, java.sql.Time, java.sql.Timestamp, oracle.sql.TIMESTAMPLTZ, java.lang.String, byte[]

Table A–1 (Cont.) Valid SQL Data Type-Java Class Mappings

These SQL data types:	Can be materialized as these Java types:
Object types	`oracle.sql.STRUCT`
	`java.sql.Struct`
	`java.sql.SqlData`
	`oracle.sql.ORAData`
Reference types	`oracle.sql.REF`
	`java.sql.Ref`
	`oracle.sql.ORAData`
Nested table types and `VARRAY` types	`oracle.sql.ARRAY`
	`java.sql.Array`
	`oracle.sql.ORAData`

Notes:

- 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 the 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 (see Note)
NCHAR VARYING	no
NUMBER	yes
NVARCHAR2	no (see Note)
RAW	yes
REF	yes

Table A–2 (Cont.) Support for SQL Data Types

SQL Data Type	Supported by JDBC Drivers?
ROWID	yes
UROWID	no
VARCHAR2	yes

Note: 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.

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

Table A–4 (Cont.) Support for SQL User-Defined Types

SQL User-Defined type	Supported by JDBC Drivers?
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 `BOOLEAN` and `DATE` data types
- 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)

Table A–5 (Cont.) Support for PL/SQL Data Types

PL/SQL Data Type	Supported by JDBC Drivers?
RAW	yes
ROWID	yes
STRING	yes
UROWID	no
VARCHAR	yes
VARCHAR2	yes
BOOLEAN	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

Notes:

- 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 21-2 for details.

Embedded SQL92 Syntax

The Oracle JDBC drivers support some embedded SQL92 syntax, which is the syntax that you specify between curly braces. The current support is basic. This section describes the support offered by the drivers for the following SQL92 constructs:

- Time and Date Literals
- Scalar Functions

- LIKE Escape Characters
- Outer Joins
- Function Call Syntax

Where driver support is limited, these sections also describe possible workarounds.

Disabling Escape Processing

Escape processing for SQL92 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 SQL92 syntax and escape 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("scott");
ods.setPassword("tiger");
Connection conn = ods.getConnection();

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

// Select the ename column from the emp table where the hiredate is Jan-23-1982
ResultSet rset = stmt.executeQuery
                 ("SELECT ename FROM emp WHERE hiredate = {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 ename FROM emp WHERE hiredate = {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 ename FROM emp WHERE hiredate = {ts '1982-01-23 12:00:00'}");
```

Scalar Functions

The 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 '\\'}");
```

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 a, dept b 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,...) }
```

SQL92 to SQL Syntax Example

You can write a simple program to translate SQL92 syntax to standard SQL syntax. The following program prints the comparable SQL syntax for SQL92 statements for function calls, date literals, time literals, and timestamp literals. In the program, the `oracle.jdbc.OracleSql` class `parse()` method performs the conversions.

```
import oracle.jdbc.OracleSql;

public class Foo
{
   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 + " => " +
         oracle.jdbc.OracleDriver.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_DATE ('1998-10-22 16:22:34', 'YYYY-MM-DD
HH24:MI:SS')
```

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
- SQL92 Outer Join Escapes
- PL/SQL TABLE, BOOLEAN, and RECORD Types
- IEEE 754 Floating Point Compliance
- Catalog Arguments to DatabaseMetaData Calls
- SQLWarning Class
- 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 5-14 for more information on how to use and manipulate `ROWID`s.

SQL92 Outer Join Escapes

Oracle JDBC drivers do not support SQL92 outer join escapes. Use Oracle SQL syntax with + instead.

> **See Also:** "Embedded SQL92 Syntax" on page A-6

PL/SQL TABLE, BOOLEAN, and RECORD Types

It is not feasible for Oracle JDBC drivers to support calling arguments or return values of the PL/SQL `RECORD`, `BOOLEAN`, or table with non-scalar element types. However, Oracle JDBC drivers support PL/SQL index-by table of scalar element types.

> **See Also:** "Accessing PL/SQL Index-by Tables" on page 5-4

As a workaround to PL/SQL `RECORD`, `BOOLEAN`, or non-scalar table types, create wrapper procedures that handle the data as types supported by JDBC. For example, to wrap a stored procedure that uses PL/SQL boolean, create a stored procedure that takes a character or number from JDBC and passes it to the original procedure as `BOOLEAN` or, for an output parameter, accepts a `BOOLEAN` argument from the original procedure and passes it as a `CHAR` or `NUMBER` to JDBC. Similarly, to wrap a stored procedure that uses PL/SQL records, create a stored procedure that handles a record in its individual components, such as `CHAR` and `NUMBER`, or in a structured object type. To wrap a stored procedure that uses PL/SQL tables, break the data into components or perhaps use Oracle collection types.

> **See Also:** "Boolean Parameters in PL/SQL Stored Procedures" on page D-2 for an example of a workaround for `BOOLEAN`.

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:** "DatabaseMetaData TABLE_REMARKS Reporting" on page 25-20 for information on how the Oracle JDBC drivers treat the `catalog` argument.

SQLWarning Class

The `java.sql.SQLWarning` class provides information on 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. The 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 3-7

Binding Named Parameters

Binding by name is not supported when using the `set`*XXX* methods. Under certain circumstances, previous versions of the 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. In 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 5-21 and "Interface oracle.jdbc.OraclePreparedStatement" on page 5-20

Retaining Bound Values

Before Oracle9*i* Database, the Oracle JDBC drivers did not retain bound values from one call of `execute` to the next as specified in JDBC 1.0. All releases after Oracle9*i* Database have retained bound values. For example:

```
PreparedStatement p = conn.prepareStatement
    ("SELECT name FROM emp WHERE id = ? AND dept = ?");
p.setInt(1, 314159);
p.setString(2, "SALES");
ResultSet r1 = p.execute();
p.setInt(1, 425260);
ResultSet r2 = p.execute();
```

Previously, a `SQLException` would be thrown by the second `execute` call because no value was bound to the second argument. In Oracle Database 10*g*, the second `execute` will return the correct value, retaining the binding of the second argument to the string `SALES`.

If the retained bound value is a stream, then the Oracle JDBC drivers will not reset the stream. Unless the application code resets, repositions, or otherwise modifies the stream, the subsequent calls to `execute` will send `NULL` as the value of the argument.

B

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
- Transaction Isolation Levels and Access Modes

JDBC and Multithreading

The 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.

Performance Optimization

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
- Standard and Oracle 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, the statement completes when the last row of the ResultSet has been retrieved or when the ResultSet 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 hostname after the @ sign in the connection URL.
   OracleDataSource ods = new OracleDataSource();
   ods.setURL("jdbc:oracle:oci:@");
   ods.setUser("scott");
   ods.setPassword("tiger");
   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:** "Oracle Row Prefetching" on page 25-15 and "Fetch Size" on page 19-15

Standard and Oracle Update Batching

The Oracle JDBC drivers allow 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. You can either use Oracle update batching, which typically processes a batch implicitly once a pre-set batch value is reached, or standard update batching, where the batch is processed explicitly.

See Also: "Update Batching" on page 25-1

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.

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 22, "Statement 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.

Table B–1 is a subset of the information presented in Table 13–1. The table lists the one-to-one type-mapping of the SQL database type to its Java `oracle.sql.*` representation.

Table B–1 Mapping of SQL Data Types to Java Classes that Represent SQL Data Types

SQL Data Type	ORACLE Mapping - Java Classes Representing SQL Data Types
`CHAR`	`oracle.sql.CHAR`
`VARCHAR2`	`oracle.sql.CHAR`
`DATE`	`oracle.sql.DATE`
`DECIMAL`	`oracle.sql.NUMBER`
`DOUBLE PRECISION`	`oracle.sql.NUMBER`
`FLOAT`	`oracle.sql.NUMBER`
`INTEGER`	`oracle.sql.NUMBER`
`REAL`	`oracle.sql.NUMBER`
`RAW`	`oracle.sql.RAW`
`LONG RAW`	`oracle.sql.RAW`
`REF CURSOR`	`java.sql.ResultSet`
`CLOB LOCATOR`	`oracle.sql.CLOB`
`BLOB LOCATOR`	`oracle.sql.BLOB`
`BFILE`	`oracle.sql.BFILE`
nested table	`oracle.sql.ARRAY`
varray	`oracle.sql.ARRAY`

Table B–1 (Cont.) Mapping of SQL Data Types to Java Classes that Represent SQL Data

SQL Data Type	ORACLE Mapping - Java Classes Representing SQL Data Types
SQL object value	If there is no entry for the object value in the type map: ■ `oracle.sql.STRUCT` If there is an entry for the object value in the type map: ■ customized Java class
REF to SQL object type	class that implements `oracle.sql.SQLRef`, typically by extending `oracle.sql.REF`

This mapping provides the most efficient conversion between SQL and Java data representations. It stores the usual representations of SQL data as byte arrays. It avoids re-formatting the data. It is information-preserving. This Oracle mapping is the most efficient type-mapping for applications that "shovel" data from SQL to Java, or vice versa.

The most efficient way to access numeric data is as 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.

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. In Oracle Database 10*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

Read-only connections are supported by the Oracle server, but not by the Oracle JDBC drivers.

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.

C

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 the 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 3-7

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 output 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

JDBC Messages Sorted by ORA Number

The following table lists the JDBC error messages sorted by the `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 !!
ORA-17036	exception in OracleNumber
ORA-17037	Fail to convert between UTF8 and UCS2
ORA-17038	Byte array not long enough

ORA Number	Message
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
ORA-17072	Inserted value too large for column
ORA-17073	Logical handle no longer valid
ORA-17074	invalid name pattern
ORA-17075	Invalid operation for forward only resultset

ORA Number	Message
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 combination of connections specified
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 Number	Message
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

JDBC Messages Sorted in Alphabetic Order

The following table lists the JDBC error messages sorted in alphabetic order:

ORA Number	Message
ORA-17066	Access layer specific implementation needed
ORA-17102	Bfile is read only
ORA-17038	Byte array not long enough
ORA-17084	Called on the insert row
ORA-17028	Can not do new defines until the current ResultSet is closed
ORA-17019	Can only describe a query
ORA-17078	Cannot do the operation as connections are already opened
ORA-17032	cannot set row prefetch to zero
ORA-17039	Char array not long enough
ORA-17035	Character Set Not Supported !!
ORA-17008	Closed Connection
ORA-17057	Closed LOB
ORA-17010	Closed Resultset
ORA-17009	Closed Statement
ORA-17105	connection session time zone was not set
ORA-17065	Conversion class is null

ORA Number	Message
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-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-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
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-17089	internal error
ORA-17001	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-17080	invalid batch command
ORA-17042	Invalid Batch Value
ORA-17055	Invalid character encountered in

ORA Number	Message
ORA-17003	Invalid column index
ORA-17006	Invalid column name
ORA-17004	Invalid column type
ORA-17106	invalid combination of connections specified
ORA-17124	Invalid connection cache Inactivity timeout specified
ORA-17111	Invalid connection cache TTL timeout specified
ORA-17103	invalid connection type to return via getConnection. Use getJavaSqlConnection instead
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-17074	invalid name pattern
ORA-17075	Invalid operation for forward only resultset
ORA-17076	Invalid operation for read only resultset
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-17002	Io exception
ORA-17092	JDBC statements cannot be created or executed at end of call processing
ORA-17073	Logical handle no longer valid
ORA-17033	Malformed SQL92 string at position
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 Number	Message
ORA-17056	Non supported character set
ORA-17034	Non supported SQL92 token at position
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
ORA-17012	Parameter Type Conflict
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-17104	SQL statement to execute cannot be empty or null
ORA-17109	standard Java character encoding not found
ORA-17096	Statement Caching cannot be enabled for this logical connection
ORA-17095	Statement Cache size has not been set
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-17054	The LOB locator is not valid
ORA-17053	The size is not valid
ORA-17051	This API cannot be be used for non-UDT types
ORA-17052	This ref is not valid
ORA-17113	Thread interval value is more than the cache timeout value
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-17088	Unsupported syntax for requested resultset type and concurrency level
ORA-17069	Use explicit XA call
ORA-17079	User credentials doesn't match the existing ones
ORA-17085	Value conflicts occurs

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:

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 cast 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
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:

ORA Number	Message
ORA-17203	Could not cast 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 Number	Message
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:

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
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 Number	Message
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
ORA-17443	Null user or password not supported in THIN driver
ORA-17444	TTC Protocol version received from server not supported

TTC Messages Sorted in Alphabetic Order

The following table lists the TTC messages in the 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 Number	Message
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-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-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-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

D

Troubleshooting

This appendix describes how to troubleshoot a Java Database Connectivity (JDBC) application or applet, and contains the following topics:

- Common Problems
- Basic Debugging Procedures

Common Problems

This section describes some common problems that you might encounter while using the Oracle JDBC drivers. These problems include:

- Memory Consumption for CHAR Columns Defined as OUT or IN/OUT Variables
- Memory Leaks and Running Out of Cursors
- Boolean Parameters in PL/SQL Stored Procedures
- Opening More Than 16 OCI Connections for a Process
- Using statement.cancel
- Using JDBC with Firewalls

Memory Consumption for CHAR Columns Defined as OUT or IN/OUT Variables

In PL/SQL, when a `CHAR` or a `VARCHAR` 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. Note that `VARCHAR2` columns do not exhibit this behavior.

At previous releases, the solution to the problem was to invoke the `Statement.setMaxFieldSize` method. A better solution is to use `OracleCallableStatement.registerOutParameter`.

We encourage you always to call `registerOutParameter(int paramIndex, int sqlType, int scale, int maxLength)` on each `CHAR` or `VARCHAR` column. This method is defined in `oracle.jdbc.driver.OracleCallableStatement`. Use the fourth argument, `maxLength`, to limit the memory consumption. `maxLength` tells the driver how many characters are necessary to store this column. The column will be 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. The 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.

Boolean Parameters in PL/SQL Stored Procedures

The JDBC drivers do not support the passing of `BOOLEAN` parameters to PL/SQL stored procedures. If a PL/SQL procedure contains `BOOLEAN` values, you can work around the restriction by wrapping the PL/SQL procedure with a second PL/SQL procedure that accepts the argument as an `INT` and passes it to the first stored procedure. When the second procedure is called, the server performs the conversion from `INT` to `BOOLEAN`.

The following is an example of a stored procedure, `BOOLPROC`, that attempts to pass a `BOOLEAN` parameter, and a second procedure, `BOOLWRAP`, that performs the substitution of an `INT` value for the `BOOLEAN`.

```
CREATE OR REPLACE PROCEDURE boolproc(x boolean)
AS
BEGIN
[...]
END;

CREATE OR REPLACE PROCEDURE boolwrap(x int)
AS
BEGIN
IF (x=1) THEN
  boolproc(TRUE);
ELSE
  boolproc(FALSE);
END IF;
END;

// Create the database connection from a DataSource
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci:@<...hoststring...>");
ods.setUser("scott");
ods.setPassword("tiger");
Connection conn = ods.getConnection();
CallableStatement cs = conn.prepareCall ("begin boolwrap(?); end;");
cs.setInt(1, 1);
cs.execute ();
```

Opening More Than 16 OCI Connections for a Process

You might find that you are not able 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 other thread's call to `Statement.cancel`.

As a result, `Statement.cancel` 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`. `Statement.cancel` does not free the thread that initiated the `Statement.execute`.

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 on 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 the *Oracle Database Net Services Administrator's Guide* for more information.

The JDBC standard method `Statement.setQueryTimeout` relies on `Statement.cancel`. If execution continues longer than the specified time-out interval, then the monitor thread calls `Statement.cancel`. 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 `Statement.execute`.

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.

`Statement.cancel` and `Statement.setQueryTimeout` are not supported in the server-side internal driver. The server-side internal driver runs in the single-threaded server process; 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 `Statement.cancel`. 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.net.READ_TIMEOUT` 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.

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 3-7.

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 on 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 on the internal operations of the event. This information is output 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.

TRACE_LEVEL_CLIENT

Purpose:

Turns tracing on/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.

TRACE_LEVEL_SERVER

Purpose:

Turns tracing on/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 ODBCSpy and ODBCTest.

Index

A

absolute positioning in result sets, 19-2
acceptChanges() method, 20-8
Accessing PL/SQL Index-by Tables, 5-4
addBatch() method, 25-8
APPLET HTML tag, 6-8
applets
  connecting to a database, 6-2
  deploying in an HTML page, 6-8
  packaging, 6-8
  packaging and deploying, 1-6
  signed applets
    browser security, 6-5
    object-signing certificate, 6-6
  using signed applets, 6-5
  using with firewalls, 6-6
ARCHIVE, parameter for APPLET tag, 6-9
ARRAY
  class, 5-7
  descriptors, 5-8
  objects, creating, 5-8, 18-8
array descriptor
  creating, 18-15
ArrayDescriptor object, 18-7, 18-15
  creating, 18-7
  deserialization, 18-10
  get methods, 18-9
  serialization, 18-10
  setConnection() method, 18-10
arrays
  defined, 18-1
  getting, 18-13
  named, 18-1
  passing to callable statement, 18-16
  retrieving from a result set, 18-10
  retrieving partial arrays, 18-13
  using type maps, 18-16
  working with, 18-1
authentication (security), 10-2
auto-commit, 3-5
auto-commit mode
  disabling, B-1
  result set behavior, B-2

B

batch jobs, authenticating users in, 10-7
batch updates--see update batching
batch value
  checking value, 25-5
  connection batch value, setting, 25-4
  connection vs. statement value, 25-3
  default value, 25-3
  overriding value, 25-5
  statement batch value, setting, 25-4
BatchUpdateException, 25-12
beforeFirst() method, 20-5
BFILE
  accessing data, 16-20
  class, 5-8
  creating and populating columns, 16-19
  defined, 14-6
  introduction, 16-1
  locators, 16-17
    getting from a result set, 16-17
    getting from callable statement, 16-17
    passing to callable statements, 16-17
    passing to prepared statements, 16-17
  manipulating data, 16-20
  reading data, 16-18
BFILE locator, selecting, 5-8
BigDecimal mapping (for attributes), 15-34
BLOB, 16-4
  class, 5-8
  creating and populating, 16-7
  creating columns, 16-7
  getting locators, 16-2
  interface changes, 4-8
  introduction, 16-1
  locators
    getting from result set, 16-3
    selecting, 5-8
  manipulating data, 16-8
  populating columns, 16-7
  reading data, 16-4, 16-5
  writing data, 16-6
BLOBs
  size limit with PL/SQL prodcedures, 16-4
Boolean parameters, restrictions, D-2
branch qualifier (distributed transactions), 29-13

C

CachedRowSet, 20-6
caching, client-side
  custom use for scrollable result sets, 19-4
  Oracle use for scrollable result sets, 19-4
callable statement
  getting a BFILE locator, 16-17
  getting LOB locators, 16-3
  passing BFILE locator, 16-17
  passing LOB locators, 16-4
  using getOracleObject() method, 13-7
cancelling
  SQL statements, D-3
cancelRowUpdates() method (result set), 19-12
casting return values, 13-11
catalog arguments (DatabaseMetaData), A-11
CHAR class
  conversions with KPRB driver, 8-6
CHAR columns
  space padding, D-1
  using setFixedCHAR() to match in WHERE, 13-16
character sets, 5-13
  conversions with KPRB driver, 8-6
checksums
  code example, 10-5
  setting parameters in Java, 10-5
  support by OCI drivers, 10-3
  support by Thin driver, 10-4
CLASSPATH, specifying, 2-3
clearBatch() method, 25-10
clearClientIdentifier() method, 5-19
clearDefines() method, 25-17
CLOB
  class, 5-8
  creating and populating, 16-7
  creating columns, 16-7
  interface changes, 4-8
  introduction, 16-1
  locators, 16-2
    getting from result set, 16-3
    passing to callable statements, 16-4
    passing to prepared statement, 16-4
  locators, selecting, 5-8
  manipulating data, 16-8
  populating columns, 16-7
  reading data, 16-4, 16-6
  writing data, 16-6
close(), 22-3
close() method, 5-20, 5-21, 5-23, D-2
  for caching statements, 22-5, 22-6
closeFile() method, 16-21
closeWithKey(), 22-3
closeWithKey() method, 22-7
CMAN.ORA file, creating, 6-4
CODE, parameter for APPLET tag, 6-8
CODEBASE, parameter for APPLET tag, 6-9
collections
  defined, 18-1
collections (nested tables and arrays), 18-7
column types
  defining, 25-17
  redefining, 25-14
commit a distributed transaction branch, 29-12
commit changes to database, 3-5
CONCUR_READ_ONLY result sets, 19-5
CONCUR_UPDATABLE result sets, 19-5
concurrency types in result sets, 19-3
CONNECT / feature, 10-7
connect string
  Connection Manager, 6-5
connection
  closing, 3-5
  from KPRB driver, 1-7
  opening, 3-2
connection attributes, 23-6
connection cache properties, 23-8
Connection Manager, 6-3
  installing, 6-4
  starting, 6-4
  using, 6-3
  using multiple managers, 6-5
  writing the connect string, 6-5
connection methods, JDBC 2.0 result sets, 19-20
connection properties, 9-7
  put() method, 9-12
connections
  read-only, B-4
constants for SQL types, 5-24
CREATE DIRECTORY statement
  for BFILEs, 16-19
CREATE TABLE statement
  to create BFILE columns, 16-19
  to create BLOB, CLOB columns, 16-7
CREATE TYPE statement, 15-21
create() method
  for ORADataFactory interface, 15-16
createDescriptor() method, 15-5, 18-9
createStatement(), 22-3
createStatement() method, 5-19, 22-7
createTemporary() method, 16-15
CursorName
  limitations, A-11
cursors, D-2
custom collection classes
  and JPublisher, 18-18
  defined, 18-1, 18-18
custom Java classes, 5-3
  defined, 15-1
custom object classes
  creating, 15-8
  defined, 15-1
custom reference classes
  and JPublisher, 17-5
  defined, 17-1, 17-5

D

data conversions, 13-3
  LONG, 14-2

LONG RAW, 14-2
data sources
creating and connecting (with JNDI), 9-6
creating and connecting (without JNDI), 9-6
logging and tracing, 9-12
Oracle implementation, 9-2
PrintWriter, 9-12
properties, 9-2
standard interface, 9-2
data streaming
avoiding, 14-5
data type classes, 5-5
data type mappings, 13-1
data types
Java, 13-1
Java native, 13-1
JDBC, 13-1
Oracle SQL, 13-1
database
connecting
from an applet, 6-2
via multiple Connection Managers, 6-5
with server-side internal driver, 8-1
connection testing, 2-5
database meta data methods, JDBC 2.0 result sets, 19-22
database specifiers, 9-13
database URL
including userid and password, 3-3
database URL, specifying, 3-2
database URLs
and database specifiers, 9-13
DatabaseMetaData calls, A-11
DatabaseMetaData class, A-8
datasources, 9-1
and JNDI, 9-6 to 9-7
DATE class, 5-8
DBMS_SERVICE.SERVICE_TIME, 24-2
DBMS_SERVICE.THROUGHPUT, 24-2
debugging JDBC programs, D-4
DEFAULT_CHARSET character set value, 5-13
defaultConnection() method, 8-1
defineColumnType() method, 5-20, 14-5, 25-17
DELETE in a result set, 19-11
deleteRow() method (result set), 19-11
deletesAreDetected() method (database meta data), 19-18
deserialization
ArrayDescriptor object, 18-10
creating a StructDescriptor object, 15-7
creating an ArrayDescriptor object, 18-10
definition of, 15-7, 18-10
StructDescriptor object, 15-7
distributed transaction ID component, 29-13
distributed transactions
branch qualifier, 29-13
check for same resource manager, 29-13
commit a transaction branch, 29-12
components and scenarios, 29-2
concepts, 29-2
distributed transaction ID component, 29-13
end a transaction branch, 29-10
example of implementation, 29-15
global transaction identifier, 29-13
ID format identifier, 29-13
introduction, 29-1
Oracle XA connection implementation, 29-6
Oracle XA data source implementation, 29-6
Oracle XA ID implementation, 29-13
Oracle XA optimizations, 29-15
Oracle XA resource implementation, 29-7
prepare a transaction branch, 29-11
roll back a transaction branch, 29-12
start a transaction branch, 29-8
transaction branch ID component, 29-13
XA connection interface, 29-6
XA data source interface, 29-6
XA error handling, 29-15
XA exception classes, 29-14
XA ID interface, 29-13
XA resource functionality, 29-8
XA resource interface, 29-7
DML Returning, 5-4, 5-26
example, 5-28
limitations, 5-29
Oracle-specific APIs, 5-27
running statements, 5-27
DMS
and end-to-end matrics, 30-1
Double.NaN
restrictions on use, 5-8
driverType, 9-4

E

enabling SSL, 11-5
encryption
code example, 10-5
overview, 10-2
setting parameters in Java, 10-5
support by OCI drivers, 10-3
support by Thin driver, 10-4
end a distributed transaction branch, 29-10
end-to-end matrics
and DMS, 30-1
end-to-end metrics, 30-1 to 30-3
Enterprise Java Beans (EJB), 20-8
environment variables
specifying, 2-3
errors
general JDBC message structure, C-1
general JDBC messages, listed, C-1
processing exceptions, 3-7
TTC messages, listed, C-10
exceptions
retrieving error code, 3-8
retrieving message, 3-8
retrieving SQL state, 3-8
execute() method, 20-9
executeBatch() method, 25-9

executeQuery() method, 5-19
executeUpdate() method, 25-6
explicit statement caching
  definition of, 22-2
  null data, 22-7
extensions to JDBC, Oracle, 5-1, 13-1, 15-1, 17-1, 18-1, 25-1
external changes (result set)
  defined, 19-17
  seeing, 19-18
  visibility vs. detection, 19-18
external file
  defined, 14-6

F

failover
  fast connection, 27-1 to 27-8
Fast Connection Failover, 27-1 to 27-8
fast connection failover
  prerequisites, 27-2
fetch direction in result sets, 19-10
fetch size, result sets, 19-15
FilteredRowSet, 20-11
finalizer methods, D-2
firewalls
  configuring for applets, 6-6
  connect string, 6-7
  described, 6-6
  required rule list items, 6-6
  using with applets, 1-6, 6-6
Firewalls, using with JDBC, D-3
floating-point compliance, A-11
Float.NaN
  restrictions on use, 5-8
format identifier, transaction ID, 29-13
forward-only result sets, 19-2
freeTemporary() method, 16-15
function call syntax, SQL92 syntax, A-9

G

getARRAY() method, 18-10
getArray() method, 18-4, 18-7, 18-11
  using type maps, 18-12
getArrayType() method, 18-9
getAsciiStream() method, 16-11, 16-12
  for reading CLOB data, 16-5
getAttributes() method
  used by Structs, 15-12
getAutoBuffering() method
  of the oracle.sql.ARRAY class, 18-6
  of the oracle.sql.STRUCT class, 15-7
getBaseName() method, 18-9
getBaseType() method, 18-4, 18-9, 18-13
getBaseTypeName() method, 17-2, 18-4
getBinaryStream() method, 14-3, 16-21
  for reading BFILE data, 16-18
  for reading BLOB data, 16-5
getBufferSize() method, 16-9, 16-11
getBytes() method, 5-6, 14-4, 16-9, 16-21
getCallWithKey(), 22-3
getCallWithKey() method, 22-7, 22-8
getCharacterStream() method, 16-11, 16-12
  for reading CLOB data, 16-5
getChars() method, 16-11
getChunkSize() method, 16-10, 16-11
getColumnCount() method, 13-17
getColumnName() method, 13-17
getColumns() method, 25-20
getColumnType() method, 13-17
getColumnTypeName() method, 13-17
getConcurrency() method (result set), 19-7
getConnection() method, 8-1, 18-9, 26-6
getCursor() method, 5-15, 5-16
getCursorName() method
  limitations, A-11
getDefaultExecuteBatch() method, 5-19, 25-5
getDefaultRowPrefetch() method, 5-19, 25-16
getDescriptor() method, 15-3, 18-4
getDirAlias() method, 16-20, 16-21
getErrorCode() method (SQLException), 3-8
getExecuteBatch() method, 5-20, 25-4, 25-5
getFetchSize() method, 19-15
getJavaSQLConnection() method, 15-3, 18-5
getJavaSqlConnection() method, 5-25
getMaxLength() method, 18-9
getMessage() method (SQLException), 3-8
getName() method, 16-20, 16-21
getNumericFunctions() method, A-8
getObject() method
  casting return values, 13-11
  for object references, 17-3
  for ORAData objects, 15-17
  for SQLInput streams, 15-12
  for SQLOutput streams, 15-13
  for Struct objects, 15-4
  return types, 13-6, 13-7
  to get BFILE locators, 16-17
  to get Oracle objects, 15-4
  used with ORAData interface, 15-18
getOracleArray() method, 18-4, 18-11, 18-13
getOracleAttributes() method, 15-3, 15-4
getOracleObject() method, 5-21, 5-23
  casting return values, 13-11
  return types, 13-6, 13-7
  using in callable statement, 13-7
  using in result set, 13-7
getOraclePlsqlIndexTable() method, 5-30, 5-33
  argument
    int paramIndex, 5-34
  code example, 5-34
getORAData() method, 15-17, 15-18
getPassword() method, 9-3
getPlsqlIndexTable() method, 5-30, 5-32, 5-33, 5-34
  arguments
    Class primitiveType, 5-34
    int paramIndex, 5-34
  code example, 5-33, 5-35
getProcedureColumns() method, 25-20

getProcedures() method, 25-20
getREF() method, 17-4
getResultSet() method, 5-20, 18-4
getRow() method (result set), 19-9
getRowPrefetch() method, 5-20, 25-15
getSQLState() method (SQLException), 3-8
getSQLTypeName() method, 15-3, 18-4, 18-13
getStatementCacheSize() method
  code example, 22-5
getStatementWithKey(), 22-3
getStatementWithKey() method, 22-7, 22-8
getString() method, 5-13
  to get ROWIDs, 5-14
getStringFunctions() method, A-8
getStringWithReplacement() method, 5-13
getSTRUCT() method, 15-4
getSubString() method, 16-11
  for reading CLOB data, 16-5
getSystemFunctions() method, A-8
getTimeDateFunctions() method, A-8
getTransactionIsolation() method, 5-19, B-4
getType() method (result set), 19-7
getTypeMap() method, 5-19, 15-10
getUpdateCounts() method
  (BatchUpdateException), 25-12
getValue() method, 17-3
  for object references, 17-3
getXXX() methods
  casting return values, 13-11
  for specific data types, 13-9
  Oracle extended properties, 9-5
global transaction identifier (distributed
  transactions), 29-13
global transactions, 29-1
globalization, 21-1 to ??
  using, 21-1

H

HEIGHT, parameter for APPLET tag, 6-8
HTML tags, to deploy applets, 6-8

I

IEEE 754 floating-point compliance, A-11
implicit connection cache, 23-1
  example, 23-5
implicit statement caching
  definition of, 22-2
  Least Recently Used (LRU) scheme, 22-2
IN OUT parameter mode, 5-32
IN parameter mode, 5-30
INSERT in a result set, 19-13
INSERT INTO statement
  for creating BFILE columns, 16-19
insertRow() method (result set), 19-14
insertsAreDetected() method (database meta
  data), 19-18
installation
  directories and files, 2-2
  verifying on the client, 2-1
Instant Client feature, 7-2
integrity
  code example, 10-5
  overview, 10-2
  setting parameters in Java, 10-5
  support by OCI drivers, 10-3
  support by Thin driver, 10-4
internal changes (result set)
  defined, 19-17
  seeing, 19-17
isAfterLast() method (result set), 19-9
isBeforeFirst() method (result set), 19-9
isFileOpen() method, 16-21
isFirst() method (result set), 19-9
isLast() method (result set), 19-9
isSameRM() (distributed transactions), 29-13
isTemporary() method, 16-15

J

Java
  compiling and running, 2-4
  data types, 13-1
  native data types, 13-1
  stored procedures, 3-7
  stream data, 14-1
Java Naming and Directory Interface (JNDI), 9-1
Java Sockets, 1-3
Java virtual machine (JVM), 8-1
java.math, Java math packages, 3-2
java.sql, JDBC packages, 3-2
java.sql.SQLException() method, 3-7
java.sql.Struct class
  getSQLTypeName() method, 15-3
java.sql.Types class, 25-18
java.util.Map class, 18-13
java.util.Properties, 26-5
JDBC
  and IDEs, 1-7
  basic program, 3-1
  data types, 13-1
  defined, 1-1
  importing packages, 3-2
  limitations of Oracle extensions, A-10
  sample files, 2-4
  testing, 2-5
  version support, 4-1 to 4-8
JDBC 2.0 support
  data type support, 4-2
  extended feature support, 4-2
  introduction, 4-1
  JDK 1.2.x vs. JDK 1.1.x, 4-1, 4-2
  overview of features, 4-3
  standard feature support, 4-2
JDBC drivers
  applets, 1-6
  choosing a driver for your needs, 1-4
  common features, 1-2
  common problems, D-1

determining driver version, 2-4
introduction, 1-1
restrictions, D-2
SQL92 syntax, A-6
JDBC mapping (for attributes), 15-33
JdbcCheckup program, 2-5
JDBCRowSet, 20-8
JDBCSpy, D-6
JDBCTest, D-6
JDeveloper, 1-7
JDK
versions supported, 1-7
JNDI
and datasources, 9-6 to 9-7
looking up data source, 9-7
overview of Oracle support, 9-1
registering data source, 9-7
JoinRowSet, 20-13
JPublisher, 15-19, 15-32
JPublisher utility, 15-8
creating custom collection classes, 18-18
creating custom Java classes, 15-32
creating custom reference classes, 17-5
SQL type categories and mapping options, 15-33
type mapping modes and settings, 15-33
type mappings, 15-33
JSSE properties, 11-4
example, 11-4
JVM, 8-1

K

KPRB driver
overview, 1-4
relation to the SQL engine, 8-1
session context, 8-4
testing, 8-4
transaction context, 8-4
URL for, 8-2

L

LD_LIBRARY_PATH variable, specifying, 2-3
LDAP
and SSL, 9-15
Least Recently Used (LRU) scheme, 22-2, 26-5
length() method, 16-10, 16-11, 16-21, 18-5
libheteroxa10_g.so shared library, 29-21
libheteroxa10.so shared library, 29-21
LIKE escape characters, SQL92 syntax, A-9
limitations on setBytes() and setString(), use of streams to avoid, 14-9
Load Balancing Advisory, 24-2
LOB
defined, 14-6
introduction, 16-1
locators, 16-2
reading data, 16-4
LOB locators
getting from callable statements, 16-3
passing, 16-3
LOBs
empty, 16-12
new interface methods, 4-8
locators
getting for BFILEs, 16-17
getting for BLOBs, 16-2
getting for CLOBs, 16-2
LOB, 16-2
passing to callable statements, 16-4
passing to prepared statement, 16-4
logging with a data source, 9-12
LONG
data conversions, 14-2
LONG RAW
data conversions, 14-2
LRU scheme, 22-2, 26-5

M

make() method, 5-12
memory leaks, D-2
metrics
end-to-end, 30-1 to 30-3
moveToCurrentRow() method (result set), 19-13
moveToInsertRow() method (result set), 19-13
mutable arrays, 18-18

N

named arrays, 18-1
defined, 18-7
nativeXA, 9-4, 29-20
network events, trapping, D-4
next() method, 20-5
next() method (result set), 19-10
NLS. See globalization
NLS_LANG variable
desupported, 21-1
non-JSSE properties, 11-2
example, 11-3
NULL
testing for, 13-4
NULL data
converting, 13-4
null data
explicit statement caching, 22-7
NullPointerException
thrown when converting Double.NaN and Float.NaN, 5-8
NUMBER class, 5-8

O

object references
accessing object values, 17-3, 17-5
described, 17-1
passing to prepared statements, 17-4
retrieving, 17-3
retrieving from callable statement, 17-4
updating object values, 17-3, 17-5

object-JDBC mapping (for attributes), 15-33
OCI driver
  described, 1-3
ODBCSpy, D-6
ODBCTest, D-6
ONS
  configuring, 27-2 to 27-4
ons.config file, 27-2, 27-3, 27-4
openFile() method, 16-22
optimization, performance, B-1
Oracle Advanced Security
  support by JDBC, 10-1
  support by OCI drivers, 10-1
  support by Thin driver, 10-2
Oracle Connection Manager, 1-6, 6-3
Oracle data types
  using, 13-1
Oracle extensions, 5-1
  data type support, 5-2
  limitations, A-10
    catalog arguments to DatabaseMetaData calls, A-11
    CursorName, A-11
    IEEE 754 floating-point compliance, A-11
    PL/SQL TABLE, BOOLEAN, RECORD types, A-11
    read-only connection, B-4
    SQL92 outer join escapes, A-11
    SQLWarning class, A-12
  object support, 5-2
  result sets, 13-5
  statements, 13-5
  to JDBC, 5-1, 13-1, 15-1, 17-1, 18-1, 25-1
Oracle JPublisher, 5-3
  generated classes, 15-27
Oracle mapping (for attributes), 15-33
Oracle Notification Service. See ONS
Oracle objects
  and JDBC, 15-1
  converting with ORAData interface, 15-16
  converting with SQLData interface, 15-12
  getting with getObject() method, 15-4
  Java classes which support, 15-2
  mapping to custom object classes, 15-8
  reading data by using SQLData interface, 15-13
  working with, 15-1
  writing data by using SQLData interface, 15-15
Oracle SQL data types, 13-1
OracleCallableStatement interface, 5-21
  getOraclePlsqlIndexTable() method, 5-30
  getPlsqlIndexTable() method, 5-30
  getXXX() methods, 13-9
  registerIndexTableOutParameter() method, 5-30, 5-32
  registerOutParameter() method, 13-13
  setPlsqlIndexTable() method, 5-30
OracleCallableStatement object, 22-2
OracleConnection class, 5-18
OracleConnection interface, 26-2
OracleConnection object, 22-1
OracleDatabaseMetaData class, A-8
OracleDataSource class, 9-2, 26-2
oracle.jdbc. package, 5-17
oracle.jdbc., Oracle JDBC extensions, 3-2
oracle.jdbc.OracleCallableStatement interface, 5-21
  close() method, 5-23
  getOracleObject() method, 5-21
  getXXX() methods, 5-21, 5-24
  registerOutParameter() method, 5-23
  setNull() method, 5-22
  setOracleObject() methods, 5-22
  setXXX() methods, 5-22
oracle.jdbc.OracleConnection interface, 5-18
  clearClientIdentifier() method, 5-19
  createStatement() method, 5-19
  getDefaultExecuteBatch() method, 5-19
  getDefaultRowPrefetch() method, 5-19
  getTransactionIsolation() method, 5-19, B-4
  getTypeMap() method, 5-19
  prepareCall() method, 5-19
  prepareStatement() method, 5-19
  setClientIdentifier() method, 5-19
  setDefaultExecuteBatch() method, 5-19
  setDefaultRowPrefetch() method, 5-19
  setTransactionIsolation() method, 5-19, B-4
  setTypeMap() method, 5-19
oracle.jdbc.OraclePreparedStatement interface, 5-20
  close() method, 5-21
  getExecuteBatch() method, 5-20
  setExecuteBatch() method, 5-20
  setNull() method, 5-21
  setOracleObject() method, 5-20
  setORAData() method, 5-21
  setXXX() methods, 5-20
oracle.jdbc.OracleResultSet, 13-5
oracle.jdbc.OracleResultSet interface, 5-23
  getOracleObject() method, 5-23
oracle.jdbc.OracleResultSetMetaData interface, 5-24, 13-17
  getColumnCount() method, 13-17
  getColumnName() method, 13-17
  getColumnType() method, 13-17
  getColumnTypeName() method, 13-17
  using, 13-17
oracle.jdbc.OracleSql class, A-10
oracle.jdbc.OracleStatement, 13-5
oracle.jdbc.OracleStatement interface, 5-19
  close() method, 5-20
  defineColumnType(), 5-20
  executeQuery() method, 5-19
  getResultSet() method, 5-20
  getRowPrefetch() method, 5-20
  setRowPrefetch() method, 5-20
oracle.jdbc.OracleTypes class, 5-24, 25-18
oracle.jdbc.pool package, 26-3
oracle.jdbc.xa package and subpackages, 29-5
OracleOCIConnection class, 26-2
OracleOCIConnectionPool class, 26-1, 26-2
OraclePreparedStatement interface, 5-20
  getOraclePlsqlIndexTable() method, 5-30

getPlsqlIndexTable() method, 5-30
registerIndexTableOutParameter() method, 5-30
setPlsqlIndexTable() method, 5-30
OraclePreparedStatement object, 22-2
OracleResultSet interface, 5-23
getXXX() methods, 13-9
OracleResultSetCache interface, 19-4
OracleResultSetMetaData interface, 5-24
OracleServerDriver class
defaultConnection() method, 8-2
oracle.sql data type classes, 5-5
oracle.sql package
data conversions, 13-3
described, 5-4
oracle.sql.ARRAY class, 18-1
and nested tables, 5-7
and VARRAYs, 5-7
createDescriptor() method, 18-9
getArray() method, 18-4
getArrayType() method, 18-9
getAutoBuffering() method, 18-6
getBaseType() method, 18-4
getBaseTypeName() method, 18-4
getDescriptor() method, 18-4
getJavaSQLConnection() method, 18-5, 18-9
getMaxLength() method, 18-9
getOracleArray() method, 18-4
getResultSet() method, 18-4
getSQLTypeName() method, 18-4
length() method, 18-5
methods for Java primitive types, 18-5
setAutoBuffering() method, 18-6
setAutoIndexing() method, 18-6
oracle.sql.ArrayDescriptor class
getBaseName() method, 18-9
getBaseType() method, 18-9
oracle.sql.BFILE class, 5-8
closeFile() method, 16-21
getBinaryStream() method, 16-21
getBytes() method, 16-21
getDirAlias() method, 16-21
getName() method, 16-21
isFileOpen() method, 16-21
length() method, 16-21
openFile() method, 16-22
position() method, 16-22
oracle.sql.BLOB class, 5-8
getBufferSize() method, 16-9
getBytes() method, 16-9
getChunkSize() method, 16-10
length() method, 16-10
position() method, 16-10
putBytes() method, 16-10
setBinaryStream() method, 16-10
oracle.sql.CHAR class, 8-6
getString() method, 5-13
getStringWithReplacement() method, 5-13
toString() method, 5-13
oracle.sql.CharacterSet class, 5-12
oracle.sql.CLOB class, 5-8
getAsciiStream() method, 16-11
getBufferSize() method, 16-11
getCharacterStream() method, 16-11
getChars() method, 16-11
getChunkSize() method, 16-11
getSubString() method, 16-11
length() method, 16-11
position() method, 16-11
putChars() method, 16-12
setAsciiStream() method, 16-12
setCharacterStream() method, 16-12
setString() method, 16-12
supported character sets, 16-9
oracle.sql.data types
support, 5-6
oracle.sql.DATE class, 5-8
oracle.sql.Datum array, 5-34
oracle.sql.Datum class, described, 5-5
oracle.sql.NUMBER class, 5-8
oracle.sql.ORAData interface, 15-16
oracle.sql.ORADataFactory interface, 15-16
OracleSql.parse() method, A-10
oracle.sql.RAW class, 5-8
oracle.sql.REF class, 5-7, 17-1
getBaseTypeName() method, 17-2
getValue() method, 17-3
setValue() method, 17-3
oracle.sql.ROWID class, 5-6, 5-9, 5-14
oracle.sql.STRUCT class, 5-6, 15-3
getAutoBuffering() method, 15-7
getDescriptor() method, 15-3
getJavaSQLConnection() method, 15-3
getOracleAttributes() method, 15-3
setAutoBuffering() method, 15-7
toJDBC() method, 15-3
oracle.sql.StructDescriptor class
createDescriptor() method, 15-5
OracleStatement interface, 5-19
OracleTypes class, 5-24
OracleTypes class for typecodes, 5-24
OracleTypes.CURSOR variable, 5-16
OracleXAConnection class, 29-6
OracleXADataSource class, 29-6
OracleXAResource class, 29-7, 29-8
OracleXid class, 29-13
ORAData interface, 5-3
additional uses, 15-20
advantages, 15-9
Oracle object types, 15-1
reading data, 15-18
writing data, 15-19
orai18n.jar file, 21-2
othersDeletesAreVisible() method (database meta data), 19-18
othersInsertsAreVisible() method (database meta data), 19-18
othersUpdatesAreVisible() method (database meta data), 19-18
OUT parameter mode, 5-31, 5-32
outer joins, SQL92 syntax, A-9

ownDeletesAreVisible() method (database meta deta), 19-17
ownInsertsAreVisible() method (database meta data), 19-17
ownUpdatesAreVisible() method (database meta data), 19-17

P

parameter modes
  IN, 5-30
  IN OUT, 5-32
  OUT, 5-31, 5-32
password, specifying, 3-2
PATH variable, specifying, 2-4
PDA, 20-8
performance enhancements, standard vs. Oracle, 4-2
performance extensions
  defining column types, 25-17
  prefetching rows, 25-15
  TABLE_REMARKS reporting, 25-20
performance optimization, B-1
Personal Digital Assistant (PDA), 20-8
PL/SQL
  limit on BLOB size, 16-4
  restrictions, D-2
  space padding, D-1
  stored procedures, 3-6
PL/SQL index-by tables, 5-29
  mapping, 5-32
  scalar data types, 5-29
PL/SQL types
  corresponding JDBC types, 5-29
  limitations, A-11
PoolConfig() method, 26-5
populate() method, 20-7
position() method, 16-10, 16-11, 16-22
positioning in result sets, 19-2
prefetching rows, 25-14, 25-15
  suggested default, 25-17
prepare a distributed transaction branch, 29-11
prepareCall(), 22-3
prepareCall() method, 5-19, 22-5, 22-6, 22-7
prepared statement
  passing BFILE locator, 16-17
  passing LOB locators, 16-4
PreparedStatement object
  creating, 3-4
prepareStatement(), 22-3
prepareStatement() method, 5-19, 22-5, 22-6, 22-7
  code example, 22-6
previous() method (result set), 19-10
PrintWriter for a data source, 9-12
put() method
  for Properties object, 9-12
  for type maps, 15-10, 15-11
putBytes() method, 16-10
putChars() method, 16-12

Q

query, executing, 3-3

R

racgons, 27-4
RAW class, 5-8
read-only result set concurrency type, 19-3
readSQL() method, 15-12
  implementing, 15-12
REF class, 5-7
REF CURSORs, 5-15
  materialized as result set objects, 5-15
refetching rows into a result set, 19-16, 19-18
refreshRow() method (result set), 19-16
registerIndexTableOutParameter() method, 5-30, 5-32
  arguments
    int elemMaxLen, 5-32
    int elemSqlType, 5-32
    int maxLen, 5-32
    int paramIndex, 5-32
  code example, 5-32
registerOutParameter() method, 5-23, 13-13
relative positioning in result sets, 19-2
remarksReporting flag, 25-15
Remote Method Invocation (RMI), 20-7
resource managers, 29-2
result set
  auto-commit mode, B-2
  getting BFILE locators, 16-17
  getting LOB locators, 16-3
  metadata, 5-24
  Oracle extensions, 13-5
  using getOracleObject() method, 13-7
result set enhancemennts
  positioning result sets, 19-8
result set enhancements
  concurrency types, 19-3
  downgrade rules, 19-7
  fetch size, 19-15
  limitations, 19-6
  Oracle scrollability requirements, 19-4
  Oracle updatability requirements, 19-4
  positioning, 19-2
  processing result sets, 19-10
  refetching rows, 19-16, 19-18
  result set types, 19-2
  scrollability, 19-2
  seeing external changes, 19-18
  seeing internal changes, 19-17
  sensitivity to database changes, 19-2
  specifying scrollability, updatability, 19-5
  summary of methods, 19-20
  summary of visibility of changes, 19-19
  updatability, 19-3
  updating result sets, 19-11
  visibility vs. detection of external changes, 19-18
result set fetch size, 19-15
Result Set Holdability, 4-8

result set methods, JDBC 2.0, 19-20
result set object
  closing, 3-4
result set types for scrollability and sensitivity, 19-2
result set, processing, 3-3
ResultSet class, 3-3
ResultSet() method, 18-7
Retrieval of Auto-Generated Keys, 4-6
return types
  for getXXX() methods, 13-9
  getObject() method, 13-7
  getOracleObject() method, 13-7
return values
  casting, 13-11
RMI, 20-7
roll back a distributed transaction branch, 29-12
roll back changes to database, 3-5
row prefetching, 25-15
  and data streams, 14-9
ROWID class, 5-9
  CursorName methods, A-11
  defined, 5-14
ROWID, use for result set updates, 19-4
RowSet
  events and event listeners, 20-2
  overview, 20-1
  properties, 20-2
  traversing, 20-4
  types, 20-1
run-time connection load balancing, 24-1
  enabling, 24-2
  how it works, 24-1
  Load Balancing Advisory, 24-2
  overview, 24-1

S

savepoints
  transaction, 4-3 to 4-6
scalar functions, SQL92 syntax, A-8
Schema Naming, 5-3
scripts, authenticating users in, 10-7
scrollability in result sets, 19-2
scrollable result sets
  creating, 19-5
  fetch direction, 19-10
  implementation of scroll-sensitivity, 19-19
  positioning, 19-8
  processing backward/forward, 19-10
  refetching rows, 19-16, 19-18
  scroll-insensitive result sets, 19-2
  scroll-sensitive result sets, 19-2
  seeing external changes, 19-18
  visibility vs. detection of external changes, 19-18
scroll-sensitive result sets
  limitations, 19-6
security
  authentication, 10-2
  encryption, 10-2
  integrity, 10-2
  Oracle Advanced Security support, 10-1
SELECT statement
  to retrieve object references, 17-3
  to select LOB locator, 16-8
sendBatch() method, 25-5, 25-6
sensitivity in result sets to database changes, 19-2
serialization
  ArrayDescriptor object, 18-10
  definition of, 15-6, 18-10
  StructDescriptor object, 15-7
server-side internal driver
  connection to database, 8-1
server-side Thin driver, overview, 1-3
session context, 1-6
  for KPRB driver, 8-4
setAsciiStream() method, 13-15
  for writing CLOB data, 16-5
setAutoBuffering() method
  of the oracle.sql.ARRAY class, 18-6
  of the oracle.sql.STRUCT class, 15-7
setAutoCommit() method, B-1
setAutoIndexing() method, 18-6
  direction parameter values
    ARRAY.ACCESS_FORWARD, 18-6
    ARRAY.ACCESS_REVERSE, 18-6
    ARRAY.ACCESS_UNKNOWN, 18-6
setBFILE() method, 16-18
setBinaryStream() method, 13-15, 16-10
  for writing BLOB data, 16-5
setBLOB() method, 16-4
setBytes() limitations, using streams to avoid, 14-9
setCharacterStream() method, 13-16
  for writing CLOB data, 16-5
setClientIdentifier() method, 5-19
setCLOB() method, 16-4
setConnection() method
  ArrayDescriptor object, 18-10
  StructDescriptor object, 15-7
setCursorName() method, A-11
setDate() method, 13-16
setDefaultExecuteBatch() method, 5-19, 25-4
setDefaultRowPrefetch() method, 5-19, 25-16
setDisableStatementCaching() method, 22-6
setEscapeProcessing() method, A-7
setExecuteBatch() method, 5-20, 25-4
setFetchSize() method, 19-15
setFixedCHAR() method, 13-16
setFormOfUse() method, 5-11
setMaxFieldSize() method, 25-18
setNull(), 13-5
setNull() method, 5-21, 5-22, 13-13
setObejct() method, 13-12
setObject() method
  for BFILES, 16-17
  for CustomDatum objects, 15-17
  for object references, 17-4
  for STRUCT objects, 15-7
  to write object data, 15-19
setOracleObject() method, 5-20, 5-22, 13-12
  for BFILES, 16-18

for BLOBs and CLOBs, 16-4
setORAData() method, 5-21, 15-17, 15-19
setPlsqlIndexTable() method, 5-30
arguments
int curLen, 5-31
int elemMaxLen, 5-31
int elemSqlType, 5-31
int maxLen, 5-31
int paramIndex, 5-30, 5-33
Object arrayData, 5-30
code example, 5-31
setPoolConfig() method, 26-4
setREF() method, 17-4
setRemarksReporting() method, 25-20
setResultSetCache() method, 19-5
setRowPrefetch() method, 5-20, 25-15
setString() limitations, using streams to avoid, 14-9
setString() method, 16-12
to bind ROWIDs, 5-14
setTime() method, 13-16
setTimestamp() method, 13-16
setTransactionIsolation() method, 5-19, B-4
setTypeMap() method, 5-19
setUnicodeStream() method, 13-16
setValue() method, 17-3
setXXX() methods
Oracle extended properties, 9-5
setXXX() methods, for empty LOBs, 16-12
setXXX() methods, for specific data types, 13-13
signed applets, 1-6
Solaris
shared libraries, 29-21
specifiers
database, 9-13
SQL
data converting to Java data types, 13-3
types, constants for, 5-24
SQL engine
relation to the KPRB driver, 8-1
SQL syntax (Oracle), A-7
SQL92 syntax, A-6
function call syntax, A-9
LIKE escape characters, A-9
outer joins, A-9
scalar functions, A-8
time and date literals, A-7
translating to SQL example, A-10
SQLData interface, 5-3
advantages, 15-9
described, 15-12
Oracle object types, 15-1
reading data from Oracle objects, 15-13
using with type map, 15-12
writing data from Oracle objects, 15-15
SQLInput interface, 15-12
described, 15-12
SQLInput streams, 15-12
SQLNET.ORA
parameters for tracing, D-4
SQLOutput interface, 15-12
described, 15-12
SQLOutput streams, 15-13
SQLWarning class, limitations, A-12
SSL
and LDAP, 9-15
SSL in JDBC, 11-2
SSL Support, 11-1
SSL, enabling, 11-5
start a distributed transaction branch, 29-8
statement caching
explicit
definition of, 22-2
null data, 22-7
implicit
definition of, 22-2
Least Recently Used (LRU) scheme, 22-2
statement methods, JDBC 2.0 result sets, 19-22
Statement object
closing, 3-4
creating, 3-3
statement.cancel(), D-3
statements
Oracle extensions, 13-5
stopping
statement execution, D-3
stored procedures
Java, 3-7
PL/SQL, 3-6
stream data, 14-1, 16-4
CHAR columns, 14-5
closing, 14-8
example, 14-3
external files, 14-6
LOBs, 14-6
LONG columns, 14-2
LONG RAW columns, 14-2
multiple columns, 14-7
precautions, 14-8
RAW columns, 14-5
row prefetching, 14-9
UPDATE/COMMIT statements, 16-5
use to avoid setBytes() and setString() limitations, 14-9
VARCHAR columns, 14-5
stream data column
bypassing, 14-7
STRUCT class, 5-6
STRUCT descriptor, 15-5, 15-6
STRUCT object, 5-6
attributes, 5-7
creating, 15-5
embedded object, 15-4
nested objects, 5-7
retrieving, 15-4
retrieving attributes as oracle.sql types, 15-4
StructDescriptor object
creating, 15-5
deserialization, 15-7
get methods, 15-6
serialization, 15-7

setConnection() method, 15-7

T

TABLE_REMARKS columns, 25-15
TABLE_REMARKS reporting
  restrictions on, 25-20
TAF, definition of, 28-1
TCP/IP protocol, 9-14
testing
  for NULL values, 13-4
Thin driver
  applications, 1-5
  LDAP over SSL, 9-15
  overview, 1-3
  server-side, overview, 1-3
time and date literals, SQL92 syntax, A-7
tnsEntry, 9-4, 29-20
toDatum() method
  applied to CustomDatum objects, 15-9, 15-16
  called by setORAData() method, 15-19
toJDBC() method, 15-3
toJdbc() method, 5-6
toString() method, 5-13
trace facility, D-4
trace parameters
  client-side, D-4
  server-side, D-6
tracing with a data source, 9-12
transaction branch, 29-1
transaction branch ID component, 29-13
transaction context, 1-6
  for KPRB driver, 8-4
transaction IDs (distributed transactions), 29-3
transaction managers, 29-2
transaction savepoints, 4-3 to 4-6
transactions
  switching between local and global, 29-4 to 29-5
Transparent Application Failover (TAF), definition of, 28-1
TTC error messages, listed, C-10
type map, 5-3, 13-6
  adding entries, 15-10
  and STRUCTs, 15-11
  creating a new map, 15-11
  used with arrays, 18-12
  used with SQLData interface, 15-12
  using with arrays, 18-16
type map (SQL to Java), 15-8
type mapping
  BigDecimal mapping, 15-34
  JDBC mapping, 15-33
  object JDBC mapping, 15-33
  Oracle mapping, 15-33
type mappings
  JPublisher options, 15-33
type maps
  relationship to database connection, 8-3
TYPE_FORWARD_ONLY result sets, 19-5
TYPE_SCROLL_INSENSITIVE result sets, 19-5
TYPE_SCROLL_SENSITIVE result sets, 19-5
typecodes, Oracle extensions, 5-24

U

unicode data, 5-10
updatability in result sets, 19-3
updatable result set concurrency type, 19-3
updatable result sets
  creating, 19-5
  DELETE operations, 19-11
  INSERT operations, 19-13
  limitations, 19-6
  refetching rows, 19-16, 19-18
  seeing internal changes, 19-17
  update conflicts, 19-14
  UPDATE operations, 19-12
update batching
  overview, Oracle vs. standard model, 25-2
  overview, statements supported, 25-2
update batching (Oracle model)
  batch value, checking, 25-5
  batch value, overriding, 25-5
  committing changes, 25-6
  connection batch value, setting, 25-4
  connection vs. statement batch value, 25-3
  default batch value, 25-3
  disable auto-commit, 25-3
  example, 25-7
  limitations and characteristics, 25-3
  overview, 25-3
  statement batch value, setting, 25-4
  stream types not allowed, 25-3
  update counts, 25-6
update batching (standard model)
  adding to batch, 25-8
  clearing the batch, 25-10
  committing changes, 25-10
  error handling, 25-12
  example, 25-11
  executing the batch, 25-9
  intermixing batched and non-batched, 25-13
  overview, 25-8
  stream types not allowed, 25-8
  update counts, 25-11
  update counts upon error, 25-12
update conflicts in result sets, 19-14
update counts
  Oracle update batching, 25-6
  standard update batching, 25-11
  upon error (standard batching), 25-12
UPDATE in a result set, 19-12
updateRow() method (result set), 19-12
updatesAreDetected() method (database meta data), 19-18
updateXXX() methods (result set), 19-12, 19-13
updateXXX() methods for empty LOBs, 16-12
updating result sets, 19-11
url, 9-4
URLs

for KPRB driver, 8-2
userid, specifying, 3-2

V

VARCHAR2 columns, D-1

W

WebRowSet, 20-9
WIDTH, parameter for APPLET tag, 6-8
window, scroll-sensitive result sets, 19-19
writeSQL() method, 15-12, 15-13
implementing, 15-12

X

XA
connection implementation, 29-6
connections (definition), 29-3
data source implementation, 29-6
data sources (definition), 29-2
definition, 29-2
error handling, 29-15
example of implementation, 29-15
exception classes, 29-14
Oracle optimizations, 29-15
Oracle transaction ID implementation, 29-13
resource implementation, 29-7
resources (definition), 29-3
transaction ID interface, 29-13
XAException, 29-12
Xids, 29-12