

Oracle9i

JDBC Developer's Guide and Reference

Release 1 (9.0.1)

June 2001

Part No. A90211-01

ORACLE®

Contributors: Magdi Morsi, Ron Peterson, Ekkehard Rohwedder, Ashok Shivarudraiah, Catherine Wong, Ed Shirk, Sunil Kunisetty, Joyce Yang, Mehul Bastawala, Luxi Chidambaran, Srinath Krishnaswamy, Rajkumar Irudayaraj, Scott Urman, Jerry Schwarz, Steve Ding, Soulaïman Htite, Douglas Surber, Anthony Lai, Paul Lo, Prabha Krishna, Ellen Barnes, Susan Kraft, Sheryl Maring, Angie Long

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and JDeveloper, Oracle Net, Oracle Objects, Oracle9i, Oracle8i, Oracle8, Oracle7, PL/SQL, SQL*Net, and SQL*Plus are trademarks or registered trademarks of Oracle Corporation. Other names may be trademarks of their respective owners.

Contents

Send Us Your Comments	xix
Preface.....	xxi
Intended Audience	xxi
Structure.....	xxii
Conventions.....	xxiv
Related Documents.....	xxiv
Documentation Accessibility	xxvii
1 Overview	
Introduction	1-2
What is JDBC?	1-2
JDBC versus SQLJ.....	1-2
Overview of the Oracle JDBC Drivers.....	1-5
Common Features of Oracle JDBC Drivers	1-6
JDBC Thin Driver.....	1-7
JDBC OCI Driver	1-8
JDBC Server-Side Thin Driver	1-9
JDBC Server-Side Internal Driver	1-10
Choosing the Appropriate Driver	1-10
Overview of Application and Applet Functionality.....	1-12
Application Basics	1-12
Applet Basics	1-12
Oracle Extensions	1-13

Package oracle.jdbc.....	1-13
Server-Side Basics	1-15
Session and Transaction Context.....	1-15
Connecting to the Database.....	1-15
Environments and Support	1-16
Supported JDK and JDBC Versions.....	1-16
JNI and Java Environments.....	1-16
JDBC and the Oracle Application Server.....	1-17
JDBC and IDEs.....	1-17

2 Getting Started

Requirements and Compatibilities for Oracle JDBC Drivers	2-2
Verifying a JDBC Client Installation	2-5
Check Installed Directories and Files.....	2-5
Check the Environment Variables.....	2-6
Make Sure You Can Compile and Run Java.....	2-7
Determine the Version of the JDBC Driver.....	2-8
Testing JDBC and the Database Connection: JdbcCheckup.....	2-8

3 Basic Features

First Steps in JDBC	3-2
Import Packages.....	3-2
Register the JDBC Drivers.....	3-3
Open a Connection to a Database.....	3-3
Create a Statement Object.....	3-11
Execute a Query and Return a Result Set Object.....	3-11
Process the Result Set.....	3-11
Close the Result Set and Statement Objects.....	3-12
Make Changes to the Database.....	3-12
Commit Changes.....	3-13
Close the Connection.....	3-14
Sample: Connecting, Querying, and Processing the Results	3-15
Datatype Mappings	3-16
Table of Mappings.....	3-16
Notes Regarding Mappings.....	3-18

Java Streams in JDBC	3-19
Streaming LONG or LONG RAW Columns	3-19
Streaming CHAR, VARCHAR, or RAW Columns.....	3-24
Data Streaming and Multiple Columns	3-25
Streaming LOBs and External Files	3-27
Closing a Stream.....	3-28
Notes and Precautions on Streams	3-28
Stored Procedure Calls in JDBC Programs	3-31
PL/SQL Stored Procedures.....	3-31
Java Stored Procedures	3-32
Processing SQL Exceptions	3-33
Retrieving Error Information.....	3-33
Printing the Stack Trace.....	3-34
4 Overview of JDBC 2.0 Support	
Introduction	4-2
JDBC 2.0 Support: JDK 1.2.x versus JDK 1.1.x	4-3
Datatype Support	4-3
Standard Feature Support	4-4
Extended Feature Support	4-5
Standard versus Oracle Performance Enhancement APIs	4-5
Migration from JDK 1.1.x to JDK 1.2.x	4-5
Overview of JDBC 2.0 Features	4-7
5 Overview of Oracle Extensions	
Introduction to Oracle Extensions	5-2
Support Features of the Oracle Extensions	5-3
Support for Oracle Datatypes	5-3
Support for Oracle Objects.....	5-4
Support for Schema Naming	5-5
OCI Extensions.....	5-6
Oracle JDBC Packages and Classes	5-7
Package oracle.sql.....	5-7
Package oracle.jdbc.....	5-16
Package oracle.jdbc2 (for JDK 1.1.x only)	5-26

Oracle Character Datatypes Support	5-28
SQL CHAR Datatypes.....	5-28
SQL NCHAR Datatypes	5-28
Class oracle.sql.CHAR.....	5-29
Additional Oracle Type Extensions	5-33
Oracle ROWID Type	5-33
Oracle REF CURSOR Type Category.....	5-34
Support for Oracle Extensions in 8.0.x and 7.3.x JDBC Drivers.....	5-36

6 Accessing and Manipulating Oracle Data

Data Conversion Considerations	6-2
Standard Types versus Oracle Types.....	6-2
Converting SQL NULL Data.....	6-2
Result Set and Statement Extensions	6-3
Comparison of Oracle get and set Methods to Standard JDBC	6-4
Standard getObject() Method.....	6-4
Oracle getOracleObject() Method.....	6-4
Summary of getObject() and getOracleObject() Return Types	6-6
Other getXXX() Methods	6-7
Casting Your get Method Return Values	6-10
Standard setObject() and Oracle setOracleObject() Methods.....	6-11
Other setXXX() Methods.....	6-12
Limitations of the Oracle 8.0.x and 7.3.x JDBC Drivers	6-18
Using Result Set Meta Data Extensions	6-19

7 Working with LOBs and BFILES

Oracle Extensions for LOBs and BFILES	7-2
Working with BLOBs and CLOBs	7-3
Getting and Passing BLOB and CLOB Locators	7-3
Reading and Writing BLOB and CLOB Data	7-6
Creating and Populating a BLOB or CLOB Column.....	7-10
Accessing and Manipulating BLOB and CLOB Data	7-12
Additional BLOB and CLOB Features.....	7-13
Working With Temporary LOBs	7-18
Using Open and Close With LOBs	7-19

Working with BFILES	7-20
Getting and Passing BFILE Locators	7-20
Reading BFILE Data	7-22
Creating and Populating a BFILE Column.....	7-23
Accessing and Manipulating BFILE Data	7-25
Additional BFILE Features.....	7-26

8 Working with Oracle Object Types

Mapping Oracle Objects	8-2
Using the Default STRUCT Class for Oracle Objects	8-3
STRUCT Class Functionality	8-3
Creating STRUCT Objects and Descriptors.....	8-4
Retrieving STRUCT Objects and Attributes	8-7
Binding STRUCT Objects into Statements.....	8-8
STRUCT Automatic Attribute Buffering.....	8-9
Creating and Using Custom Object Classes for Oracle Objects	8-10
Relative Advantages of ORADData versus SQLData.....	8-11
Understanding Type Maps for SQLData Implementations.....	8-11
Creating a Type Map Object and Defining Mappings for a SQLData Implementation ..	8-12
Understanding the SQLData Interface	8-15
Reading and Writing Data with a SQLData Implementation.....	8-17
Understanding the ORADData Interface.....	8-21
Reading and Writing Data with a ORADData Implementation.....	8-23
Additional Uses for ORADData	8-26
The Deprecated CustomDatum Interface	8-28
Object-Type Inheritance	8-29
Creating Subtypes	8-29
Implementing Customized Classes for Subtypes	8-30
Retrieving Subtype Objects.....	8-37
Creating Subtype Objects	8-40
Sending Subtype Objects.....	8-41
Accessing Subtype Data Fields.....	8-41
Inheritance Meta Data Methods.....	8-43
Using JPublisher to Create Custom Object Classes	8-45
JPublisher Functionality	8-45

JPublisher Type Mappings	8-45
Describing an Object Type	8-49
Functionality for Getting Object Meta Data.....	8-49
Steps for Retrieving Object Meta Data.....	8-50
SQLJ Object Types	8-52
Creating a SQLJ Object Type in SQL Representation.....	8-53
Inserting an Instance of a SQLJ Object Type.....	8-59
Retrieving Instances of a SQLJ Object Type.....	8-60
Meta Data Methods for SQLJ Object Types	8-61
SQLJ Object Types and Custom Object Types Compared.....	8-62

9 Working with Oracle Object References

Oracle Extensions for Object References	9-2
Overview of Object Reference Functionality	9-4
Object Reference Getter and Setter Methods	9-4
Key REF Class Methods.....	9-4
Retrieving and Passing an Object Reference	9-6
Retrieving an Object Reference from a Result Set.....	9-6
Retrieving an Object Reference from a Callable Statement	9-7
Passing an Object Reference to a Prepared Statement	9-8
Accessing and Updating Object Values through an Object Reference	9-9
Custom Reference Classes with JPublisher	9-10

10 Working with Oracle Collections

Oracle Extensions for Collections (Arrays)	10-2
Choices in Materializing Collections	10-2
Creating Collections	10-3
Creating Multi-Level Collection Types	10-4
Overview of Collection (Array) Functionality	10-5
Array Getter and Setter Methods	10-5
ARRAY Descriptors and ARRAY Class Functionality.....	10-6
ARRAY Performance Extension Methods	10-8
Accessing oracle.sql.ARRAY Elements as Arrays of Java Primitive Types	10-8
ARRAY Automatic Element Buffering.....	10-9
ARRAY Automatic Indexing	10-9

Creating and Using Arrays	10-11
Creating ARRAY Objects and Descriptors	10-11
Retrieving an Array and Its Elements	10-15
Passing Arrays to Statement Objects	10-22
Using a Type Map to Map Array Elements	10-25
Custom Collection Classes with JPublisher	10-27

11 Result Set Enhancements

Overview	11-2
Result Set Functionality and Result Set Categories Supported in JDBC 2.0.....	11-2
Oracle JDBC Implementation Overview for Result Set Enhancements	11-5
Creating Scrollable or Updatable Result Sets	11-8
Specifying Result Set Scrollability and Updatability	11-8
Result Set Limitations and Downgrade Rules.....	11-10
Positioning and Processing in Scrollable Result Sets	11-13
Positioning in a Scrollable Result Set.....	11-13
Processing a Scrollable Result Set	11-15
Updating Result Sets	11-18
Performing a DELETE Operation in a Result Set	11-18
Performing an UPDATE Operation in a Result Set.....	11-19
Performing an INSERT Operation in a Result Set	11-21
Update Conflicts	11-23
Fetch Size	11-24
Setting the Fetch Size	11-24
Use of Standard Fetch Size versus Oracle Row-Prefetch Setting	11-25
Refetching Rows	11-26
Seeing Database Changes Made Internally and Externally	11-27
Seeing Internal Changes	11-27
Seeing External Changes	11-28
Visibility versus Detection of External Changes.....	11-29
Summary of Visibility of Internal and External Changes.....	11-30
Oracle Implementation of Scroll-Sensitive Result Sets	11-30
Summary of New Methods for Result Set Enhancements	11-32
Modified Connection Methods.....	11-32
New Result Set Methods	11-32

New Statement Methods	11-35
New Database Meta Data Methods	11-35

12 Performance Extensions

Update Batching	12-2
Overview of Update Batching Models	12-2
Oracle Update Batching	12-4
Standard Update Batching	12-10
Premature Batch Flush	12-18
Additional Oracle Performance Extensions	12-20
Oracle Row Prefetching	12-20
Defining Column Types	12-23
DatabaseMetaData TABLE_REMARKS Reporting	12-26

13 Statement Caching

About Statement Caching	13-2
Basics of Statement Caching	13-2
Implicit Statement Caching	13-2
Explicit Statement Caching	13-4
Comparing Implicit and Explicit Statement Caching	13-4
Using Statement Caching	13-6
Enabling and Disabling Statement Caching	13-6
Checking for Statement Creation Status	13-7
Physically Closing a Cached Statement	13-7
Using Implicit Statement Caching	13-8
Using Explicit Statement Caching	13-9

14 Connection Pooling and Caching

Data Sources	14-2
A Brief Overview of Oracle Data Source Support for JNDI	14-2
Data Source Features and Properties	14-3
Creating a Data Source Instance and Connecting (without JNDI)	14-7
Creating a Data Source Instance, Registering with JNDI, and Connecting	14-8
Logging and Tracing	14-11

Connection Pooling	14-13
Connection Pooling Concepts.....	14-13
Connection Pool Data Source Interface and Oracle Implementation	14-14
Pooled Connection Interface and Oracle Implementation	14-15
Creating a Connection Pool Data Source and Connecting.....	14-16
Connection Caching	14-18
Overview of Connection Caching	14-18
Typical Steps in Using a Connection Cache	14-21
Oracle Connection Cache Specification: OracleConnectionCache Interface	14-24
Oracle Connection Cache Implementation: OracleConnectionCacheImpl Class	14-25
Oracle Connection Event Listener: OracleConnectionEventListener Class.....	14-29
15 Distributed Transactions	
Overview	15-2
Distributed Transaction Components and Scenarios.....	15-2
Distributed Transaction Concepts.....	15-3
Oracle XA Packages	15-5
XA Components	15-6
XA Data Source Interface and Oracle Implementation.....	15-6
XA Connection Interface and Oracle Implementation.....	15-7
XA Resource Interface and Oracle Implementation	15-8
XA Resource Method Functionality and Input Parameters	15-9
XA ID Interface and Oracle Implementation.....	15-13
Error Handling and Optimizations	15-15
XA Exception Classes and Methods	15-15
Mapping between Oracle Errors and XA Errors.....	15-16
XA Error Handling	15-16
Oracle XA Optimizations	15-17
Implementing a Distributed Transaction	15-18
Summary of Imports for Oracle XA.....	15-18
Oracle XA Code Sample	15-18
16 JDBC OCI Extensions	
OCI Driver Connection Pooling	16-2
OCI Driver Connection Pooling: Background.....	16-3

OCI Driver Connection Pooling and MTS Compared	16-3
Stateless Sessions Compared to Stateful Sessions.....	16-4
Defining an OCI Connection Pool.....	16-4
Connecting to an OCI Connection Pool	16-9
Statement Handling and Caching	16-10
JNDI and the OCI Connection Pool	16-12
Middle-Tier Authentication Through Proxy Connections	16-13
OCI Driver Transparent Application Failover	16-16
Failover Type Events.....	16-16
TAF Callbacks	16-17
Java TAF Callback Interface	16-17
OCI HeteroRM XA.....	16-19
Configuration and Installation	16-19
Exception Handling.....	16-19
HeteroRM XA Code Example.....	16-19
Accessing PL/SQL Index-by Tables.....	16-21
Overview.....	16-21
Binding IN Parameters.....	16-22
Receiving OUT Parameters	16-24

17 Java Transaction API

Transaction Overview	17-2
Global and Local Transactions.....	17-2
Demarcating Transactions.....	17-3
Enlisting Resources.....	17-5
Two-Phase Commit	17-7
JTA Limitations	17-8
JTA Summary	17-9
Environment Initialization	17-9
Methods for Enlisting Database Resources.....	17-9
Summary of Single-Phase and Two-Phase Commit.....	17-10
Binding Transactional Objects in the Namespace.....	17-11
JTA Client-Side Demarcation	17-15
JTA Server-Side Database Enlistment	17-18
Configuring Two-Phase Commit Engine	17-21

Creating DataSource Objects Dynamically	17-25
JDBC Restrictions	17-26

18 Advanced Topics

JDBC and Globalization Support	18-2
How JDBC Drivers Perform Globalization Support Conversions	18-3
Globalization Support and Object Types.....	18-5
CHAR and VARCHAR2 Data Size Restrictions with the Thin Driver.....	18-6
JDBC Client-Side Security Features	18-9
JDBC Support for Oracle Advanced Security.....	18-9
JDBC Support for Login Authentication.....	18-10
JDBC Support for Data Encryption and Integrity.....	18-11
JDBC in Applets	18-16
Connecting to the Database through the Applet	18-16
Connecting to a Database on a Different Host Than the Web Server.....	18-18
Using Applets with Firewalls	18-21
Packaging Applets.....	18-24
Specifying an Applet in an HTML Page	18-25
JDBC in the Server: the Server-Side Internal Driver	18-27
Connecting to the Database with the Server-Side Internal Driver.....	18-27
Exception-Handling Extensions for the Server-Side Internal Driver.....	18-29
Session and Transaction Context for the Server-Side Internal Driver	18-31
Testing JDBC on the Server.....	18-31
Loading an Application into the Server	18-33
Server-Side Character Set Conversion of oracle.sql.CHAR Data	18-34

19 Coding Tips and Troubleshooting

JDBC and Multithreading	19-2
Performance Optimization	19-6
Disabling Auto-Commit Mode.....	19-6
Standard Fetch Size and Oracle Row Prefetching	19-7
Standard and Oracle Update Batching.....	19-7
Common Problems	19-8
Space Padding for CHAR Columns Defined as OUT or IN/OUT Variables.....	19-8
Memory Leaks and Running Out of Cursors.....	19-8

Boolean Parameters in PL/SQL Stored Procedures	19-9
Opening More Than 16 OCI Connections for a Process	19-9
Basic Debugging Procedures	19-11
Oracle Net Tracing to Trap Network Events	19-11
Third Party Debugging Tools	19-13
Transaction Isolation Levels and Access Modes	19-14

20 Sample Applications

Basic Samples	20-2
Listing Names from the EMP Table—Employee.java	20-2
Inserting Names into the EMP Table—InsertExample.java	20-3
Samples of PL/SQL in JDBC	20-6
Calling PL/SQL Stored Procedures—PLSQLExample.java	20-6
Executing Procedures in PL/SQL Blocks—PLSQL.java	20-7
Accessing a PL/SQL Index-by Table from JDBC—PLSQLIndexTab.java	20-11
Intermediate Samples	20-18
Streams—StreamExample.java	20-18
Multithreading—JdbcMTSample.java	20-20
Samples for JDBC 2.0 Types	20-25
BLOBs and CLOBs—LobExample.java	20-25
Weakly Typed Objects—PersonObject.java	20-30
Weakly Typed Object References—StudentRef.java	20-33
Weakly Typed Arrays—ArrayExample.java	20-35
Samples for Oracle Type Extensions	20-38
REF CURSORS—RefCursorExample.java	20-38
BFILES—FileExample.java	20-40
Samples for Custom Object Classes	20-43
SQLData Implementation—SQLDataExample.java	20-43
ORADATA Implementation—ORADATAExample.java	20-47
JDBC 2.0 Result Set Enhancement Samples	20-52
Positioning in a Result Set—ResultSet2.java	20-52
Inserting and Deleting Rows in a Result Set—ResultSet3.java	20-55
Updating Rows in a Result Set—ResultSet4.java	20-59
Scroll-Sensitive Result Set—ResultSet5.java	20-61
Refetching Rows in a Result Set—ResultSet6.java	20-64

Performance Enhancement Samples	20-69
Standard Update Batching—BatchUpdates.java	20-69
Oracle Update Batching with Implicit Execution—SetExecuteBatch.java	20-71
Oracle Update Batching with Explicit Execution—SendBatch.java.....	20-73
Oracle Row Prefetching Specified in Connection—RowPrefetch_connection.java.....	20-74
Oracle Row Prefetching Specified in Statement—RowPrefetch_statement.java	20-76
Oracle Column Type Definitions—DefineColumnType.java	20-78
Implicit Statement Caching—StmtCache1.java.....	20-79
Explicit Statement Caching—StmtCache2.java.....	20-82
Samples for Connection Pooling and Distributed Transactions	20-85
Data Source without JNDI—DataSource.java	20-85
Data Source with JNDI—DataSourceJNDI.java.....	20-86
Pooled Connection—PooledConnection.java.....	20-89
OCI Connection Pool—OCIConnectionPool.java.....	20-90
Middle-Tier Authentication—NtierAuth.java	20-93
JDBC OCI Application Failover Callbacks—OCIFailOver.java	20-97
Oracle Connection Cache (dynamic)—CCache1.java	20-101
Oracle Connection Cache ("fixed with no wait")—CCache2.java	20-103
XA with Suspend and Resume—XA2.java	20-105
XA with Two-Phase Commit Operation—XA4.java.....	20-109
HeteroRM XA—XA6.java.....	20-116
Sample Applet	20-120
HTML Page—JdbcApplet.htm	20-120
Applet Code—JdbcApplet.java	20-121
JDBC versus SQLJ Sample Code	20-124
SQL Program to Create Tables and Objects.....	20-124
JDBC Version of the Sample Code	20-126
SQLJ Version of the Sample Code.....	20-130

21 Reference Information

Valid SQL-JDBC Datatype Mappings	21-2
Supported SQL and PL/SQL Datatypes	21-5
Embedded SQL92 Syntax	21-10
Time and Date Literals.....	21-10
Scalar Functions	21-12

LIKE Escape Characters	21-13
Outer Joins	21-13
Function Call Syntax	21-14
SQL92 to SQL Syntax Example	21-14
Oracle JDBC Notes and Limitations	21-16
CursorName	21-16
SQL92 Outer Join Escapes	21-16
PL/SQL TABLE, BOOLEAN, and RECORD Types	21-16
IEEE 754 Floating Point Compliance	21-17
Catalog Arguments to DatabaseMetaData Calls	21-17
SQLWarning Class	21-17
Bind by Name	21-17
Related Information	21-19
Oracle JDBC Drivers and SQLJ	21-19
Java Technology	21-19

A Row Set

Introduction	A-2
Row Set Setup and Configuration	A-4
Runtime Properties for Row Set	A-5
Row Set Listener	A-6
Traversing Through the Rows	A-8
Cached Row Set	A-9
CachedRowSet Constraints	A-13
JDBC Row Set	A-15

B JDBC Error Messages

General Structure of JDBC Error Messages	B-2
General JDBC Messages	B-3
JDBC Messages Sorted by ORA Number	B-3
JDBC Messages Sorted Alphabetically	B-7
HeteroRM XA Messages	B-12
HeteroRM XA Messages Sorted by ORA Number	B-12
HeteroRM XA Messages Sorted Alphabetically	B-12
TTC Messages	B-14

TTC Messages Sorted by ORA Number.....	B-14
TTC Messages Sorted Alphabetically.....	B-15

Send Us Your Comments

Oracle9i JDBC Developer's Guide and Reference, Release 1 (9.0.1)

Part No. A90211-01

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

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

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

- Electronic mail—jpgcomment_us@oracle.com
- FAX - 650-506-7225. Attn: Java Platform Group, Information Development Manager
- Postal service:

Oracle Corporation
Java Platform Group, Information Development Manager
500 Oracle Parkway, Mailstop 4op9
Redwood Shores, CA 94065
USA

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

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

Preface

This preface introduces you to the *Oracle9i 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.

Intended Audience

This manual is intended for anyone with an interest in JDBC programming but assumes at least some prior knowledge of the following:

- Java
- SQL
- Oracle PL/SQL
- Oracle databases

Structure

The *Oracle9i JDBC Developer's Guide and Reference* contains the following chapters and appendices:

- [Chapter 1, "Overview"](#)—Provides an overview of the Oracle implementation of JDBC and the Oracle JDBC driver architecture.
- [Chapter 2, "Getting Started"](#)—Introduces the Oracle JDBC drivers and some scenarios of how you can use them. This chapter also guides you through the basics of testing your installation and configuration.
- [Chapter 3, "Basic Features"](#)—Covers the basic steps in creating any JDBC application. It also discusses additional basic features of Java and JDBC supported by the Oracle JDBC drivers.
- [Chapter 4, "Overview of JDBC 2.0 Support"](#)—Presents an overview of JDBC 2.0 features and describes the differences in how these features are supported in the JDK 1.2.x and JDK 1.1.x environments.
- [Chapter 5, "Overview of Oracle Extensions"](#)—Provides an overview of the JDBC extension classes supplied by Oracle.
- [Chapter 6, "Accessing and Manipulating Oracle Data"](#)—Describes data access using the Oracle datatype formats rather than Java formats.
- [Chapter 7, "Working with LOBs and BFILES"](#)—Covers the Oracle extensions to the JDBC standard that let you access and manipulate LOBs and LOB data.
- [Chapter 8, "Working with Oracle Object Types"](#)—Explains how to map Oracle object types to Java classes by using either standard JDBC or Oracle extensions.
- [Chapter 9, "Working with Oracle Object References"](#)—Describes the Oracle extensions to standard JDBC that let you access and manipulate object references.
- [Chapter 10, "Working with Oracle Collections"](#)—Discusses the Oracle extensions to standard JDBC that let you access and manipulate arrays and their data.
- [Chapter 11, "Result Set Enhancements"](#)—This chapter discusses JDBC 2.0 result set enhancements such as scrollable result sets and updatable result sets, including support issues under JDK 1.1.x
- [Chapter 12, "Performance Extensions"](#)—Describes Oracle extensions to the JDBC standard that enhance the performance of your applications.

- [Chapter 13, "Statement Caching"](#)—Describes Oracle extension statements for caching.
- [Chapter 14, "Connection Pooling and Caching"](#)—Discusses JDBC 2.0 data sources (and their usage of JNDI), connection pooling functionality (a framework for connection caching implementations), and a sample connection caching implementation provided by Oracle.
- [Chapter 15, "Distributed Transactions"](#)—Covers distributed transactions, otherwise known as global transactions, and standard XA functionality. (Distributed transactions are sets of transactions, often to multiple databases, that have to be committed in a coordinated manner.)
- [Chapter 16, "JDBC OCI Extensions"](#)—Describes extensions specific to the OCI driver.
- [Chapter 17, "Java Transaction API"](#)—Describes how to use the JDBC connections within a JTA global transaction in order to include all changes to multiple databases within a transaction.
- [Chapter 18, "Advanced Topics"](#)—Describes advanced JDBC topics such as using NLS, working with applets, the server-side driver, and embedded SQL92 syntax.
- [Chapter 19, "Coding Tips and Troubleshooting"](#)—Includes coding tips and general guidelines for troubleshooting your JDBC applications.
- [Chapter 20, "Sample Applications"](#)—Presents sample applications that highlight advanced JDBC features and Oracle extensions.
- [Chapter 21, "Reference Information"](#)—Contains detailed JDBC reference information.
- [Appendix A, "Row Set"](#)—Describes JDBC and cached row sets.
- [Appendix B, "JDBC Error Messages"](#)—This appendix lists JDBC error messages and the corresponding ORA error numbers.

Conventions

This book uses Solaris syntax, but file names and directory names for Windows NT are the same, unless otherwise noted.

The term [ORACLE_HOME] is used to indicate the full path of the Oracle home directory.

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

The following conventions are also used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
...	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted.
< >	Angle brackets enclose user-supplied names.
[]	Brackets enclose optional clauses from which you can choose one or none.

Related Documents

This section lists other documentation of interest.

See the following additional documents available from the Oracle Java Platform group:

- *Oracle9i Java Developer's Guide*

This book introduces the basic concepts of Java in Oracle9i 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, SQLJ, or EJBs) is in this book.

- *Oracle9i JPublisher User's Guide*

This book describes how to use the JPublisher utility to translate object types and other user-defined types to Java classes. If you are developing SQLJ or 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.

- *Oracle9i SQLJ Developer's Guide and Reference*

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

- *Oracle9i Java Stored Procedures Developer's Guide*

This book discusses Java stored procedures—programs that run directly in the Oracle9i server. With stored procedures (functions, procedures, database triggers, and SQL methods), Java developers can implement business logic at the server level, thereby improving application performance, scalability, and security.

- *Oracle9i Enterprise JavaBeans Developer's Guide and Reference*

This book describes the Oracle extensions to the Enterprise JavaBeans and CORBA specifications.

- *Oracle9i Java Tools Reference*

This book describes Oracle JVM tools and associated options.

You can also refer to the following documents from the Oracle Server Technologies group.

- *Oracle Net Services Administrator's Guide*

This book contains information about the Oracle8 Connection Manager and Oracle Net network administration in general.

- *Oracle9i Globalization and National Language Support Guide*

This book contains information about NLS environment variables, character sets, and territory and locale settings. In addition, it contains an overview of common NLS issues, typical scenarios, and NLS considerations for OCI and SQL programmers.

- *Oracle Advanced Security Administrator's Guide*

This book describes features of the Oracle Advanced Security Option (formerly known as ANO or ASO).

- *Oracle9i Application Developer's Guide - Fundamentals*

This book introduces basic design concepts and programming features in using an Oracle9i database and creating database access applications.

- *Oracle9i Application Developer's Guide - Large Objects (LOBs)*

This book describes general functionality and features of database large objects (LOBs) in Oracle9i.

- *Oracle9i Application Developer's Guide - Object-Relational Features*

This book contains general information about structured objects and other object-relational database features in Oracle9i.

- *Oracle9i Supplied PL/SQL Packages Reference*

This book documents PL/SQL packages available as part of the Oracle9i server, some of which may be useful to call from JDBC applications.

- *PL/SQL User's Guide and Reference*

PL/This book explains the concepts and features of PL/SQL, Oracle's procedural language extension to SQL.

- *Oracle9i SQL Reference*

This book contains a complete description of the content and syntax of the SQL commands and features used to manage information in an Oracle database.

- *Oracle9i Database Reference*

This book contains general reference information about the Oracle9i server.

- *Oracle9i Database Error Messages*

This book contains information about error messages that can be passed by the Oracle9i server.

Documentation from the following Oracle groups may also be of interest.

- *Oracle9i Application Server Documentation Library*

This documentation contains information about how the Oracle9i Application Server supports JDBC.

- Oracle JDeveloper online help

This documentation contains information about how the Oracle JDeveloper supports Oracle JDBC.

- *Programmer's Guide to the Oracle Call Interface*

This book describes how to develop applications using the OCI.

For documentation of SQLJ standard features and syntax, refer to ANSI NCITS 331.2-2000:

- *Information Technology - SQLJ - Part 2: SQL Types using the Java™ Programming Language* document

You can obtain this from ANSI at the following Web site:

<http://www.ansi.org/>

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. 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. 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 additional information, visit the Oracle Accessibility Program Web site at:

<http://www.oracle.com/accessibility/>

JAWS, a Windows screen reader, 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, JAWS may not always read a line of text that consists solely of a bracket or brace.

1

Overview

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

- [Introduction](#)
- [Overview of the Oracle JDBC Drivers](#)
- [Overview of Application and Applet Functionality](#)
- [Server-Side Basics](#)
- [Environments and Support](#)

Introduction

This section presents a brief introduction to Oracle JDBC, including a comparison to SQLJ.

What is JDBC?

JDBC (Java Database Connectivity) is a standard Java interface for connecting from Java to relational databases. The JDBC standard was defined by Sun Microsystems, allowing individual providers to implement and extend the standard with their own JDBC drivers.

JDBC is based on the X/Open SQL Call Level Interface and complies with the SQL92 Entry Level standard.

In addition to supporting the standard JDBC API, Oracle drivers have extensions to support Oracle-specific datatypes and to enhance performance.

JDBC versus SQLJ

This section has the following subsections:

- [Advantages of SQLJ over JDBC for Static SQL](#)
- [General Guidelines for Using JDBC and SQLJ](#)

Developers who are familiar with the Oracle Call Interface (OCI) layer of client-side C code will recognize that JDBC provides the power and flexibility for the Java programmer that OCI does for the C or C++ programmer. Just as with OCI, you can use JDBC to query and update tables where, for example, the number and types of the columns are not known until runtime. This capability is called *dynamic SQL*. Therefore, JDBC is a way to use dynamic SQL statements in Java programs. Using JDBC, a calling program can construct SQL statements at runtime. Your JDBC program is compiled and run like any other Java program. No analysis or checking of the SQL statements is performed. Any errors that are made in your SQL code raise runtime errors. JDBC is designed as an API for dynamic SQL.

However, many applications do not need to construct SQL statements dynamically because the SQL statements they use are fixed or *static*. In this case, you can use SQLJ to embed *static SQL* in Java programs. In static SQL, all the SQL statements are complete or "textually evident" in the Java program. That is, details of the database object, such as the column names, number of columns in the table, and table name, are known before runtime. SQLJ offers advantages for these applications because it permits error checking at precompile time.

The precompile step of a SQLJ program performs syntax-checking of the embedded SQL, type checking against the database to assure that the data exchanged between Java and SQL have compatible types and proper type conversions, and schema checking to assure congruence between SQL constructs and the database schema. The result of the precompilation is Java source code with SQL runtime code which, in turn, can use JDBC calls. The generated Java code compiles and runs like any other Java program.

Although SQLJ provides direct support for static SQL operations known at the time the program is written, it can also interoperate with dynamic SQL through JDBC. SQLJ allows you to create JDBC objects when they are needed for dynamic SQL operations. In this way, SQLJ and JDBC can co-exist in the same program. Convenient conversions are supported between JDBC connections and SQLJ connection contexts, as well as between JDBC result sets and SQLJ iterators. For more information on this, see the *Oracle9i SQLJ Developer's Guide and Reference*.

The syntax and semantics of SQLJ and JDBC do not depend on the configuration under which they are running, thus enabling implementation on the client or database side or in the middle tier.

Advantages of SQLJ over JDBC for Static SQL

While JDBC provides a complete dynamic SQL interface from Java to relational databases, SQLJ fills a complementary role for static SQL.

Although you can use static SQL statements in your JDBC programs, they can be represented more conveniently in SQLJ. Here are some advantages you gain in using SQLJ over JDBC for static SQL statements:

- SQLJ source programs are more concise than equivalent JDBC programs, because SQLJ provides a shorter syntax.
- SQLJ provides strong typing of connections (and the sets of SQL entities that they access), query outputs, and return parameters.
- SQLJ can use database connections to type-check static SQL code at translation time. JDBC, being a completely dynamic API, does not perform any type-checking until run-time.
- SQLJ programs allow direct embedding of Java bind expressions within SQL statements. JDBC requires separate getter and/or setter call statements for each bind variable.
- SQLJ provides simplified rules for calling SQL stored procedures and functions.

General Guidelines for Using JDBC and SQLJ

SQLJ is effective in the following circumstances:

- You want to be able to check your program for errors at translation-time, rather than at run-time.
- You want to write an application that you can deploy to another database. Using SQLJ, you can customize the static SQL for that database at deployment-time.
- You are working with a database that contains compiled SQL. You will want to use SQLJ because you cannot compile SQL statements in a JDBC program.

JDBC is effective in the following circumstances:

- Your program uses dynamic SQL. For example, you have a program that builds queries in real-time or has an interactive query component.
- You do not want to have a SQLJ layer during deployment or development. For example, you might want to download only the JDBC Thin driver and not the SQLJ runtime libraries to minimize download time over a slow link.

Note: You can intermix SQLJ code and JDBC code in the same source. This is discussed in the *Oracle9i SQLJ Developer's Guide and Reference*.

Overview of the Oracle JDBC Drivers

This section introduces the Oracle JDBC drivers, their basic architecture, and some scenarios for their use. This information describes the core functionality of all JDBC drivers. However, there is special functionality for the OCI driver, which is described [Chapter 16, "JDBC OCI Extensions"](#).

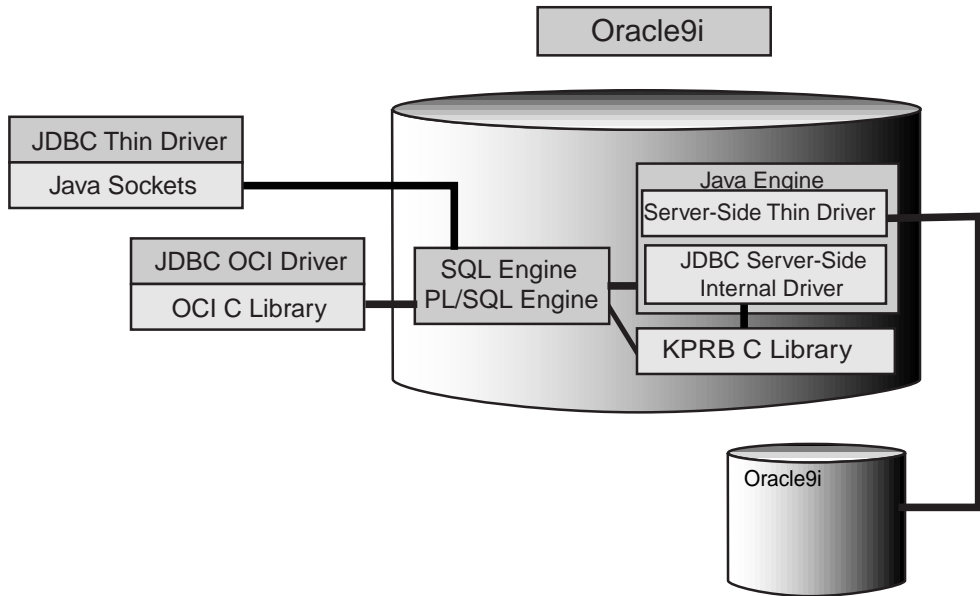
Oracle provides the following JDBC drivers:

- **Thin driver**, a 100% Java driver for client-side use without an Oracle installation, particularly with applets
- **OCI driver** for client-side use with an Oracle client installation
- **server-side Thin driver**, which is functionally the same as the client-side Thin driver, but is for code that runs inside an Oracle server and needs to access a remote server, including middle-tier scenarios
- **server-side internal driver** for code that runs inside the target server (that is, inside the Oracle server that it must access)

[Figure 1-1](#) illustrates the driver-database architecture for the JDBC Thin, OCI, and server-side internal drivers.

The rest of this section describes common features of the Oracle drivers and then discusses each one individually, concluding with a discussion of some of the considerations in choosing the appropriate driver for your application.

Figure 1–1 Driver-Database Architecture



Common Features of Oracle JDBC Drivers

The server-side and client-side Oracle JDBC drivers provide the same basic functionality. They all support the following standards and features:

- either JDK 1.2.x / JDBC 2.0 or JDK 1.1.x / JDBC 1.22 (with Oracle extensions for JDBC 2.0 functionality)

These two implementations use different sets of class files.

- same syntax and APIs
- same Oracle extensions
- full support for multi-threaded applications

Oracle JDBC drivers implement standard Sun Microsystems `java.sql` interfaces. Through the `oracle.jdbc` package, you can access the Oracle features in addition to the Sun features. This package is equivalent to the `oracle.jdbc.driver` package which is deprecated for Oracle9i.

Table 1-1 shows how the client-side drivers compare.

Table 1-1 JDBC Client-Side Drivers Compared at a Glance

Driver	Type	Size	Protocol	100% Java Use	External Libraries Needed	Platform- dependent	Perfor- mance	Comple- ness of Features	
Thin	IV	Small	TTC	Yes	Applet and application	No	No	Better	Better
OCI	II	Large	TTC	No	Application	Yes	Yes	Best	Best

Note: Most JDBC 2.0 functionality, including that for objects, arrays, and LOBs, is available in a JDK 1.1.x environment through Oracle extensions.

JDBC Thin Driver

The Oracle JDBC Thin driver is a 100% pure Java, Type IV driver. It is targeted for Oracle JDBC applets but can be used for applications as well. Because it is written entirely in Java, this driver is platform-independent. It does not require any additional Oracle software on the client side. The Thin driver communicates with the server using TTC, a protocol developed by Oracle to access the Oracle Relational Database Management System (RDBMS).

For applets it can be downloaded into a browser along with the Java applet being run. The HTTP protocol is stateless, but the Thin driver is not. The initial HTTP request to download the applet and the Thin driver is stateless. Once the Thin driver establishes the database connection, the communication between the browser and the database is stateful and in a two-tier configuration.

The JDBC Thin driver allows a direct connection to the database by providing an implementation of TCP/IP that emulates Oracle Net and TTC (the wire protocol used by OCI) 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 driver supports only TCP/IP protocol and requires a TNS listener on the TCP/IP sockets from the database server.

Note: When the JDBC Thin driver is used with an applet, the client browser must have the capability to support Java sockets.

Using the Thin driver inside an Oracle server or middle tier is considered separately, under "[JDBC Server-Side Thin Driver](#)" below.

JDBC OCI Driver

The JDBC OCI driver is a Type II driver for use with client-server Java applications. This driver requires an Oracle client installation, and therefore is Oracle platform-specific and not suitable for applets.

Note: In Oracle9*i*, the OCI driver is a single OCI driver for use with all database versions. It replaces the distinct OCI8 and OCI7 drivers of previous releases. While the OCI8 and OCI7 drivers are deprecated for Oracle9*i*, they are still supported for backward compatibility.

The JDBC OCI driver provides OCI connection pooling functionality, which can either be part of the JDBC client or a JDBC stored procedure. OCI driver connection pooling requires fewer physical connections than standard connection pooling, it also provides a uniform interface, and allows you to dynamically configure the attributes of the connection pool. For a complete description of OCI driver connection pooling, see "[OCI Driver Connection Pooling](#)" on page 16-2.

The OCI driver supports Oracle7, Oracle8/8*i*, and Oracle9*i* with the highest compatibility. It also supports all installed Oracle Net adapters, including IPC, named pipes, TCP/IP, and IPX/SPX.

The OCI driver, written in a combination of Java and C, converts JDBC invocations to calls to the Oracle Call Interface (OCI), using native methods to call C-entry points. These calls are then sent over Oracle Net to the Oracle database server. The OCI driver communicate with the server using the Oracle-developed TTC protocol.

The OCI driver uses the OCI libraries, C-entry points, Oracle Net, CORE libraries, and other necessary files on the client machine on which it is installed.

The Oracle Call Interface (OCI) is an application programming interface (API) that allows you to create applications that use the native procedures or function calls of a third-generation language to access an Oracle database server and control all phases of SQL statement execution. The OCI driver is designed to build scalable, multi-threaded applications that can support large numbers of users securely.

The Oracle9i JDBC OCI driver has the following functionality:

- Uses OCI
- Connection Pooling
- OCI optimized fetch
- Prefetching
- Fastest LOB access
- Client-side object cache
- Transparent Application Failover (TAF)
- Middle-tier authentication

JDBC Server-Side Thin Driver

The Oracle JDBC server-side Thin driver offers the same functionality as the client-side Thin driver, but runs inside an Oracle database and accesses a remote database.

This is especially useful in two situations:

- to access a remote Oracle server from an Oracle server acting as a middle tier
- more generally, to access one Oracle server from inside another, such as from any Java stored procedure or Enterprise JavaBean

There is no difference in your code between using the Thin driver from a client application or from inside a server.

Note: Statement `cancel()` and `setQueryTimeout()` methods are not supported by the server-side Thin driver.

About Permission for the Server-Side Thin Driver The thin driver opens a socket to use for its connection. Because the Oracle server is enforcing the Java security model, this means that a check is performed for a `SocketPermission` object.

To use the JDBC server-side Thin driver, the connecting user must be granted with the appropriate permission. This is an example of how the permission can be granted for 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 upper case. The '*' is a pattern. It is possible to limit the permission to allow connecting to specific machines or ports. See the Javadoc for complete details on the `java.net.SocketPermission` class. Also, refer to the *Oracle9i Java Developer's Guide* for further discussion of Java security inside the Oracle server.

JDBC Server-Side Internal Driver

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

The server-side internal driver, the JVM, the database, KPRB (server-side) C library, and the SQL engine all run within the same address space, so the issue of network round trips is irrelevant. The programs access the SQL engine by using function calls.

The server-side internal driver is fully consistent with the client-side drivers and supports the same features and extensions. For more information on the server-side internal driver, see "[JDBC in the Server: the Server-Side Internal Driver](#)" on page 18-27.

Note: The server-side internal driver supports only JDK 1.2.x.

Choosing the Appropriate Driver

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

- If you are writing an applet, you must use the JDBC Thin driver. JDBC OCI-based driver classes will not work inside a Web browser, because they call native (C language) methods.

- If you want maximum portability and performance under Oracle8i and earlier, then use the JDBC Thin driver. You can connect to an Oracle server from either an application or an applet using the JDBC Thin driver.
- If you are writing a client application for an Oracle client environment and need maximum performance, then choose the JDBC OCI driver.
- For code that runs in an Oracle server acting as a middle tier, use the server-side Thin driver.
- If your code will run inside the target Oracle server, then use the JDBC server-side internal driver to access that server. (You can also access remote servers using the server-side Thin driver.)
- If performance is critical to your application, you want maximum scalability of the Oracle server, or you need the enhanced availability features like TAF or the enhanced proxy features like middle-tier authentication, then choose the OCI driver.

Overview of Application and Applet Functionality

This section compares and contrasts the basic functionality of JDBC applications and applets, and introduces Oracle extensions that can be used by application and applet programmers.

Application Basics

You can use either the Oracle JDBC Thin or OCI driver for a client application. Because the JDBC OCI driver uses native methods, there can be significant performance advantages in using this driver for your applications.

An application that can run on a client can also run in the Oracle server, using the JDBC server-side internal driver.

If you are using a JDBC OCI driver in an application, then the application will require an Oracle installation on its clients. For example, the application will require the installation of Oracle Net and client libraries.

The JDBC Thin and OCI drivers offer support for data encryption and integrity checksum features of the Oracle Advanced Security option (formerly known as ANO or ASO). See "[JDBC Client-Side Security Features](#)" on page 18-9. Such security is not necessary for the server-side internal driver.

Applet Basics

This section describes the issues you should take into consideration if you are writing an applet that uses the JDBC Thin driver.

For more about applets and a discussion of relevant firewall, browser, and security issues, see "[JDBC in Applets](#)" on page 18-16.

Applets and Security

Without special preparations, an applet can open network connections only to the host machine from which it was downloaded. Therefore, an applet can connect to databases only on the originating machine. If you want to connect to a database running on a different machine, you have two options:

- Use Oracle8 Connection Manager on the host machine. The applet can connect to Oracle8 Connection Manager, which in turn connects to a database on another machine.
- Use signed applets, which can request socket connection privileges to other machines.

Both of these topics are described in greater detail in ["Connecting to the Database through the Applet"](#) on page 18-16.

The Thin driver offers support for data encryption and integrity checksum features of the Oracle Advanced Security option. See ["JDBC Client-Side Security Features"](#) on page 18-9.

Applets and Firewalls

An applet that uses the JDBC Thin driver can connect to a database through a firewall. See ["Using Applets with Firewalls"](#) on page 18-21 for more information on configuring the firewall and on writing connect strings for the applet.

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. This is described in detail in ["Packaging Applets"](#) on page 18-24.

Oracle Extensions

A number of Oracle extensions are available to Oracle JDBC application and applet programmers, in the following categories:

- type extensions (such as ROWIDs and REF CURSOR types)
- wrapper classes for SQL types (the `oracle.sql` package)
- support for custom Java classes to map to user-defined types
- extended LOB support
- extended connection, statement, and result set functionality
- performance enhancements

See [Chapter 5, "Overview of Oracle Extensions"](#) for an overview of type extensions and extended functionality, and succeeding chapters for further detail. See [Chapter 12, "Performance Extensions"](#) regarding Oracle performance enhancements.

Package `oracle.jdbc`

Beginning in Oracle9i, the Oracle extensions to JDBC are captured in the package `oracle.jdbc`. This package contains classes and interfaces that specify the Oracle extensions in a manner similar to the way the classes and interfaces in `java.sql` specify the public JDBC API.

Your code should use the package `oracle.jdbc` instead of the package `oracle.jdbc.driver` used in earlier versions of Oracle. Use of the package `oracle.jdbc.driver` is now deprecated, but will continue to be supported for backwards compatibility.

All that is required to convert your code is to replace "`oracle.jdbc.driver`" with "`oracle.jdbc`" in the source and recompile. This cannot be done piece-wise. You must convert all classes and interfaces that are referenced by an application. Conversion is not required, but is highly recommended. Future releases of Oracle may have features that are incompatible with use of the package `oracle.jdbc.driver`.

The purpose of this change is to enable the Oracle JDBC drivers to have multiple implementations. In all releases up to and including Oracle9i, all of the Oracle JDBC drivers have used the same top level implementation classes, the classes in the package `oracle.jdbc.driver`. By converting your code to use `oracle.jdbc`, you will be able to take advantage of future enhancements that use different implementation classes. There are no such enhancements in Oracle9i, but there are plans for such enhancements in the future.

Additionally, these interfaces permit the use of some code patterns that are difficult to use when your code uses the package `oracle.jdbc.driver`. For example, you can more easily develop wrapper classes for the Oracle JDBC classes. If you wished to wrap the `OracleStatement` class in order to log all SQL statements, you could easily do so by creating a class that wraps `OracleStatement`. That class would implement the interface `oracle.jdbc.OracleStatement` and hold an `oracle.jdbc.OracleStatement` as an instance variable. This wrapping pattern is much more difficult when your code uses the package `oracle.jdbc.driver` as you cannot extend the class `oracle.jdbc.driver.OracleStatement`.

Once again, your code should use the new package `oracle.jdbc` instead of the package `oracle.jdbc.driver`. Conversion is not required as `oracle.jdbc.driver` will continue to be supported for backwards compatibility. Conversion is highly recommended as there may in later releases be features that are not supported if your code uses `oracle.jdbc.driver`.

Server-Side Basics

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

For a complete discussion of the server-side driver, see ["JDBC in the Server: the Server-Side Internal Driver"](#) on page 18-27.

Session and Transaction Context

The server-side internal driver operates within a default session and default transaction context. For more information on default session and transaction context for the server-side driver, see ["Session and Transaction Context for the Server-Side Internal Driver"](#) on page 18-31.

Connecting to the Database

The server-side internal driver uses a default connection to the database. You can connect to the database with either the `DriverManager.getConnection()` method or the Oracle-specific `OracleDriver` class `defaultConnection()` method. For more information on connecting to the database with the server-side driver, see ["Connecting to the Database with the Server-Side Internal Driver"](#) on page 18-27.

Environments and Support

This section provides a brief discussion of platform, environment, and support features of the Oracle JDBC drivers. The following topics are discussed:

- [Supported JDK and JDBC Versions](#)
- [JNI and Java Environments](#)
- [JDBC and the Oracle Application Server](#)
- [JDBC and IDEs](#)

Supported JDK and JDBC Versions

Oracle has two versions of the Thin and OCI drivers—one that is compatible with JDK 1.2.x and one that is compatible with JDK 1.1.x. The JDK 1.2.x versions support standard JDBC 2.0. The JDK 1.1.x versions support most JDBC 2.0 features, but must do so through Oracle extensions because JDBC 2.0 features are not available in JDK 1.1.x versions.

Very little is required in migrating from a JDK 1.1.x environment to a JDK 1.2.x environment. For information, see "[Migration from JDK 1.1.x to JDK 1.2.x](#)" on page 4-5.

Notes:

- The server-side internal driver supports only JDK 1.2.x.
 - JDK 1.0.2 is no longer supported in the current drivers.
 - Each driver implementation uses its own JDBC classes ZIP file—`classes12.zip` for JDK 1.2.x versions, and `classes111.zip` for JDK 1.1.x versions.
-
-

For information about supported combinations of driver versions, JDK versions, and database versions, see "[Requirements and Compatibilities for Oracle JDBC Drivers](#)" on page 2-2.

JNI and Java Environments

The Oracle JDBC OCI driver uses the standard JNI (Java Native Interface) to call Oracle OCI C libraries. Prior to 8.1.6, when the OCI drivers supported JDK 1.0.2, they used NMI (Native Method Interface) for C calls. NMI was an earlier

specification by Sun Microsystems and was the only native call interface supported by JDK 1.0.2.

Because JNI is now supported by Oracle JDBC, you can use the OCI driver with Java virtual machines other than that of Sun Microsystems—in particular, with Microsoft and IBM JVMs. These JVMs support only JNI for native C calls.

JDBC and the Oracle Application Server

The Oracle Application Server (OAS) is a collection of middleware services and tools that provide a scalable, robust, secure, and extensible platform for distributed, object-oriented applications. The OAS supports access to applications from both Web clients (browsers) using the Hypertext Transfer Protocol (HTTP), and CORBA clients, which use the Common Object Request Broker Architecture (CORBA) and the Internet Inter-ORB Protocol (IIOP).

You can use the JDBC OCI drivers on a middle tier in conjunction with OAS (formerly Web Application Server, or WAS) versions 3.0 and higher—the OAS bundles JDBC with its distribution. For more information about the use of JDBC with the OAS, refer to the Oracle Application Server documentation.

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 Oracle Internet platform. The Oracle JDeveloper environment contains integrated support for JDBC and SQLJ, including the 100% pure JDBC Thin driver and the native OCI drivers. The database component of Oracle JDeveloper uses the JDBC drivers to manage the connection between the application running on the client and the server. See your Oracle JDeveloper documentation for more information.

Getting Started

This chapter begins by discussing compatibilities between Oracle JDBC driver versions, database versions, and JDK versions. It then guides you through the basics of testing your installation and configuration, and running a simple application. The following topics are discussed:

- [Requirements and Compatibilities for Oracle JDBC Drivers](#)
- [Verifying a JDBC Client Installation](#)

Requirements and Compatibilities for Oracle JDBC Drivers

Table 2–1 lists the compatibilities between Oracle JDBC driver versions and Oracle database versions. The JDK versions supported by each JDBC driver version are also listed.

Note: The Oracle JDBC drivers no longer support JDK 1.0.x versions.

Table 2–1 JDBC Driver-Database Compatibility

Driver Versions	Database Versions Supported	JDK Versions Supported	Drivers Available	Remarks
9.0.1	9.0.1, 8.1.7, 8.1.6, 8.1.5, 8.0.6, 8.0.5, 8.0.4, 7.3.4	1.2.x, 1.1.x	JDBC Thin driver JDBC OCI driver JDBC server-side Thin driver JDBC server-side internal driver (supports 8.1.6 database and JDK 1.2.x only)	
8.1.7	8.1.7, 8.1.6, 8.1.5, 8.0.6, 8.0.5, 8.0.4, 7.3.4	1.2.x, 1.1.x	JDBC Thin driver JDBC OCI driver JDBC server-side Thin driver JDBC server-side internal driver (supports 8.1.6 database and JDK 1.2.x only)	
8.1.6	8.1.6, 8.1.5, 8.0.6, 8.0.5, 8.0.4, 7.3.4	1.2.x, 1.1.x	JDBC Thin driver JDBC OCI driver JDBC server-side Thin driver JDBC server-side internal driver (supports 8.1.6 database and JDK 1.2.x only)	The Thin driver is also available in the server with the standard server installation. This has the same usage and functionality as the client-side Thin driver, for accessing a remote database from inside a database.
8.1.5	8.1.5, 8.0.6, 8.0.5, 8.0.4, 7.3.4	1.1.x, 1.0.x	JDBC Thin driver JDBC OCI driver JDBC server-side internal driver (supports 8.1.5 database and JDK 1.1.x only)	Both client- and server-side drivers offer full support for structured objects when run against an 8.1.5 database.

Table 2–1 JDBC Driver-Database Compatibility(Cont.)

Driver Versions	Database Versions Supported	JDK Versions Supported	Drivers Available	Remarks
8.0.6	8.0.6, 8.0.5, 8.0.4, 7.3.4	1.1.x, 1.0.x	JDBC Thin driver JDBC OCI driver <i>Note: the JDBC server-side internal driver is not available for 8.0.x and prior versions.</i>	.
8.0.5	8.0.5, 8.0.4, 7.3.4	1.1.x, 1.0.x	JDBC Thin driver JDBC OCI driver <i>Note: the JDBC server-side internal driver is not available for 8.0.x and prior versions.</i>	
8.0.4	8.0.4, 7.3.4	1.1.x, 1.0.x	JDBC Thin driver JDBC OCI driver <i>Note: the JDBC server-side internal driver is not available for 8.0.x and prior versions.</i>	
7.3.4	7.3.4	1.1.x, 1.0.x	JDBC Thin driver JDBC OCI driver <i>Note: the JDBC server-side internal driver is not available for 8.0.x and prior versions.</i>	

Notes:

- Different JDKs require different class files—classes in `classes12.zip`, `classes111.zip`, and `classes102.zip`, respectively.
 - The JDBC drivers do not support structured objects when run against an 8.0.x database. This is because JDBC depends on PL/SQL functions that did not exist in those releases.
 - There is no structured object or LOB support in Oracle 7.3.x.
 - Any client-side driver might work with 7.x databases prior to 7.3.4, but this has not been tested and is not supported.
-
-

Verifying a JDBC Client Installation

This section covers the following topics:

- [Check Installed Directories and Files](#)
- [Check the Environment Variables](#)
- [Make Sure You Can Compile and Run Java](#)
- [Determine the Version of the JDBC Driver](#)
- [Testing JDBC and the Database Connection: JdbcCheckup](#)

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. It assumes that you have already installed the driver of your choice.

If you have installed the JDBC Thin driver, no further installation on the client machine is necessary (the JDBC Thin driver requires a TCP/IP listener to be running on the database machine).

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

Check Installed Directories and Files

This section assumes that you have already installed the Sun Microsystems *Java Developer's Kit (JDK)* on your system (although other forms of Java are also supported). Oracle offers JDBC drivers compatible with either JDK 1.2.x versions or JDK 1.1.x versions.

Installing the Oracle8i Java products creates, among other things, an [ORACLE_HOME]/jdbc directory containing these subdirectories and files:

- `demo/samples`: The `samples` subdirectory 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`: The `doc` directory contains documentation about the JDBC drivers.
- `lib`: The `lib` directory contains `.zip` files with these required Java classes:

- `classes12.zip` contains the classes for use with the JDK 1.2.x—all the JDBC driver classes except the classes necessary for NLS support.
- `nls_charset12.zip` contains the classes necessary for NLS support with the JDK 1.2.x.
- `jta.zip` and `jndi.zip` contain classes for the Java Transaction API and the Java Naming and Directory Interface for JDK 1.2.x. These are only required if you will be using JTA features for distributed transaction management or JNDI features for naming services. (These files can also be obtained from the Sun Microsystems Web site, but it is advisable to use the versions from Oracle, because those have been tested with the Oracle drivers.)
- `classes111.zip` contains the classes for use with the JDK 1.1.x—all the JDBC driver classes except the classes necessary for NLS support.
`classes111.zip` also contains Oracle extensions that allow you to use JDBC 2.0 functionality for objects, arrays, and LOBs under JDK 1.1.x.
- `nls_charset11.zip` contains the classes necessary for NLS support with the JDK 1.1.x.

The `nls_charset12.zip` and `nls_charset11.zip` files provide support for specific NLS character sets. They have been separated out from the `classes*.zip` files to give you the option of excluding character sets in situations where complete NLS support is not needed. For more information on `nls_charset12.zip` and `nls_charset11.zip`, see "[Globalization Support and Object Types](#)" on page 18-5.

- `readme.txt`: The `readme.txt` file contains late-breaking and release-specific information about the drivers that might not be in this manual.

Check that all these directories have been created and populated.

Check the Environment Variables

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

You must set the `CLASSPATH` for your installed JDBC OCI or Thin driver. Depending on whether you are using the JDK 1.2.x versions or 1.1.x versions, you must set one of these values for the `CLASSPATH`:

- [Oracle Home]/jdbc/lib/classes12.zip (and optionally [Oracle Home]/jdbc/lib/nls_charset12.zip) for full NLS character support)

or:

- [Oracle Home]/jdbc/lib/classes111.zip (and optionally [Oracle Home]/jdbc/lib/nls_charset11.zip) for full NLS character support)

Ensure that there is only one `classes*.zip` file version and one `nls_charset*.zip` file version in your CLASSPATH.

Note: If you will be using JTA features or JNDI features, both of which are discussed in [Chapter 14, "Connection Pooling and Caching"](#), then you will also need to have `jta.zip` and `jndi.zip` in your CLASSPATH.

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

- On Solaris, set `LD_LIBRARY_PATH` as follows:

```
[Oracle Home]/lib
```

This directory contains the `libocijdbc9.so` shared object library.

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

```
[Oracle Home]\lib
```

This directory contains the `ocijdbc8.dll` dynamic link library.

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

Make Sure You Can Compile and Run Java

To further ensure that Java is set up properly on your client system, go to the `samples` directory (for example, `C:\oracle\ora81\jdbc\demo\samples` if you are using the JDBC driver on a Windows NT machine), then see if `javac` (the Java compiler) and `java` (the Java interpreter) will run without error. Enter:

```
javac
```

then enter:

```
java
```

Each should give you a list of options and parameters and then exit. Ideally, verify that you can compile and run a simple test program.

Determine the Version of the JDBC Driver

If at any time you must determine the version of the JDBC driver that you installed, you can invoke the `getDriverVersion()` method of the `OracleDatabaseMetaData` class.

Here is sample code showing how to do it:

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

class JDBCVersion
{
    public static void main (String args[])
        throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver
            (new oracle.jdbc.driver.OracleDriver());
        Connection conn = DriverManager.getConnection
            ("jdbc:oracle:thin:@host:port:sid", "scott", "tiger");

        // 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: JdbcCheckup

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 you for your user name, password, and the name of a database to which you want to connect. The program connects to the database, queries for the string "Hello World", and prints it to the screen.

Go to the `samples` directory and compile and run `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 executing the following:

- imports the necessary Java classes, including JDBC classes
- registers the JDBC driver
- connects to the database
- executes a simple query
- outputs the query results to your screen

"[First Steps in JDBC](#)" on page 3-2, describes these functions in greater detail. A listing of `JdbcCheckup.java` for the JDBC OCI driver appears below.

```
/*
 * 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 package to use JDBC
import java.sql.*;

// 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
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // 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...");
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:oci8:@" + database, user, password);

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 "";
}
}
```

Basic Features

This chapter covers the most basic steps taken in any JDBC application. It also describes additional basic features of Java and JDBC supported by the Oracle JDBC drivers.

The following topics are discussed:

- [First Steps in JDBC](#)
- [Sample: Connecting, Querying, and Processing the Results](#)
- [Datatype Mappings](#)
- [Java Streams in JDBC](#)
- [Stored Procedure Calls in JDBC Programs](#)
- [Processing SQL Exceptions](#)

First Steps in JDBC

This section describes how to get up and running with the Oracle JDBC drivers. 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.

To connect to and query a database from the client, you must provide code for these tasks:

1. [Import Packages](#)
2. [Register the JDBC Drivers](#)
3. [Open a Connection to a Database](#)
4. [Create a Statement Object](#)
5. [Execute a Query and Return a Result Set Object](#)
6. [Process the Result Set](#)
7. [Close the Result Set and Statement Objects](#)
8. [Make Changes to the Database](#)
9. [Commit Changes](#)
10. [Close the Connection](#)

You must supply Oracle driver-specific information for the first three tasks, which allow your program to use the JDBC API to access a database. For the other tasks, you can use standard JDBC Java code as you would for any Java application.

Import Packages

Regardless of which Oracle JDBC driver you use, include the following `import` statements at the beginning of your program (`java.math` only if needed):

```
import java.sql.*;      for standard JDBC packages
import java.math.*;    for BigDecimal and BigInteger classes
```

Import the following Oracle packages when you want to access the extended functionality provided by the Oracle drivers. However, they are not required for the example presented in this section:

```
import oracle.jdbc.*;           for Oracle extensions to JDBC
import oracle.sql.*;
```

For an overview of the Oracle extensions to the JDBC standard, see [Chapter 5, "Overview of Oracle Extensions"](#).

Register the JDBC Drivers

You must provide the code to register your installed driver with your program. You do this with the static `registerDriver()` method of the `JDBC DriverManager` class. This class provides a basic service for managing a set of JDBC drivers.

Note: Alternatively, you can use the `forName()` method of the `java.lang.Class` class to load the JDBC drivers directly. For example:

```
Class.forName ("oracle.jdbc.OracleDriver");
```

However, this method is valid only for JDK-compliant Java virtual machines. It is not valid for Microsoft Java virtual machines.

Because you are using one of Oracle's JDBC drivers, you declare a specific driver name string to `registerDriver()`. You register the driver only once in your Java application.

```
DriverManager.registerDriver (new oracle.jdbc.OracleDriver());
```

Open a Connection to a Database

Open a connection to the database with the static `getConnection()` method of the `JDBC DriverManager` class. This method returns an object of the `JDBC Connection` class that needs as input a user name, password, connect string that identifies the JDBC driver to use, and the name of the database to which you want to connect.

Connecting to a database is a step where you must enter Oracle JDBC driver-specific information in the `getConnection()` method. If you are not familiar with this method, continue reading the "[Understanding the Forms of getConnection\(\)](#)" section below.

If you are already familiar with the `getConnection()` method, you can skip ahead to either of these sections, depending on the driver you installed:

- ["Opening a Connection for the JDBC OCI Driver"](#) on page 3-9
- ["Opening a Connection for the JDBC Thin Driver"](#) on page 3-10

Notes:

- With JDK 1.2, using JNDI (Java Naming and Directory Interface) is becoming the recommended way to make connections. See ["A Brief Overview of Oracle Data Source Support for JNDI"](#) on page 14-2 and ["Creating a Data Source Instance, Registering with JNDI, and Connecting"](#) on page 14-8.
 - If you are using the Thin driver, be aware that it does not support OS authentication in making the connection. As a result, special logins are not supported.
 - This discussion in this section does not apply to the server-side internal driver, which uses an implicit connection. See ["Connecting to the Database with the Server-Side Internal Driver"](#) on page 18-27.
-
-

Understanding the Forms of `getConnection()`

The `DriverManager` class `getConnection()` method whose signatures and functionality are described in the following sections:

- ["Specifying a Database URL, User Name, and Password"](#) on page 3-5
- ["Specifying a Database URL That Includes User Name and Password"](#) on page 3-5
- ["Specifying a Database URL and Properties Object"](#) on page 3-6

If you want to specify a database name in the connection, it must be in one of the following formats:

- a Oracle Net *keyword-value* pair
- a string of the form `<host_name>:<port_number>:<sid>` (Thin driver only)
- a TNSNAMES entry (OCI driver only)

For information on how to specify a keyword-value pair or a TNSNAMES entry, see your *Oracle Net Services Administrator's Guide*.

Specifying a Database URL, User Name, and Password

The following signature takes the URL, user name, and password as separate parameters:

```
getConnection(String URL, String user, String password);
```

Where the URL is of the form:

```
jdbc:oracle:<drivertype>:@<database>
```

The following example connects user `scott` with password `tiger` to a database with SID `orcl` through port 1521 of host `myhost`, using the Thin driver.

```
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:thin:@myhost:1521:orcl", "scott", "tiger");
```

If you want to use the default connection for an OCI driver, specify either:

```
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:oci8:scott/tiger@");
```

or:

```
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:oci8:@", "scott", "tiger");
```

For all JDBC drivers, you can also specify the database with a Oracle Net keyword-value pair. The Oracle Net keyword-value pair substitutes for the `TNSNAMES` entry. The following example uses the same parameters as the preceding example, but in the keyword-value format:

```
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:oci8:@MyHostString", "scott", "tiger");
```

or:

```
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:oci8:@(description=(address=(host= myhost)
    (protocol=tcp)(port=1521))(connect_data=(sid=orcl)))", "scott", "tiger");
```

Specifying a Database URL That Includes User Name and Password

The following signature takes the URL, user name, and password all as part of a URL parameter:

```
getConnection(String URL);
```

Where the URL is of the form:

```
jdbc:oracle:<drivertype>:<user>/<password>@<database>
```

The following example connects user `scott` with password `tiger` to a database on host `myhost` using the OCI driver. In this case, however, the URL includes the `userid` and `password`, and is the only input parameter.

```
Connection conn = DriverManager.getConnection  
    ("jdbc:oracle:oci8:scott/tiger@myhost");
```

If you want to connect with the Thin driver, you must specify the port number and SID. For example, if you want to connect to the database on host `myhost` that has a TCP/IP listener up on port `1521`, and the SID (system identifier) is `orcl`:

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

Specifying a Database URL and Properties Object

The following signature takes a URL, together with a properties object that specifies user name and password (perhaps among other things):

```
getConnection(String URL, Properties info);
```

Where the URL is of the form:

```
jdbc:oracle:<drivertype>:@<database>
```

In addition to the URL, use an object of the standard Java `Properties` class as input. For example:

```
java.util.Properties info = new java.util.Properties();  
info.put ("user", "scott");  
info.put ("password", "tiger");  
info.put ("defaultRowPrefetch", "15");  
getConnection ("jdbc:oracle:oci8:@", info);
```

[Table 3-1](#) lists the connection properties that Oracle JDBC drivers support.

Table 3-1 Connection Properties Recognized by Oracle JDBC Drivers

Name	Short Name	Type	Description
user	n/a	String	the user name for logging into the database
password	n/a	String	the password for logging into the database

Table 3–1 Connection Properties Recognized by Oracle JDBC Drivers (Cont.)

Name	Short Name	Type	Description
database	server	String	the connect string for the database
internal_logon	n/a	String	a role, such as <code>sysdba</code> or <code>sysoper</code> , that allows you to log on as <code>sys</code>
defaultRowPrefetch	prefetch	String (containing integer value)	the default number of rows to prefetch from the server (default value is "10")
remarksReporting	remarks	String (containing boolean value)	"true" if <code>getTables()</code> and <code>getColumns()</code> should report <code>TABLE_REMARKS</code> ; equivalent to using <code>setRemarksReporting()</code> (default value is "false")
defaultBatchValue	batchvalue	String (containing integer value)	the default batch value that triggers an execution request (default value is "10")
includeSynonyms	synonyms	String (containing boolean value)	"true" to include column information from predefined "synonym" SQL entities when you execute a <code>DataBaseMetaData.getColumns()</code> call; equivalent to connection <code>setIncludeSynonyms()</code> call (default value is "false")

See [Table 18–4, "OCI Driver Client Parameters for Encryption and Integrity"](#) and [Table 18–5, "Thin Driver Client Parameters for Encryption and Integrity"](#) for descriptions of encryption and integrity drivers.

Using Roles for Sys Logon

To specify the role (mode) for `sys` logon, use the `internal_logon` connection property. (See [Table 3–1, "Connection Properties Recognized by Oracle JDBC Drivers"](#), for a complete description of this 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 `sys` user name.

Example The following example illustrates how to use the `internal_logon` and `sysdba` arguments to specify `sys` logon.

```
//import packages and register the driver
import java.sql.*;
import java.math.*;
DriverManager.registerDriver (new oracle.jdbc.OracleDriver());

//specify the properties object
java.util.Properties info = new java.util.Properties();
info.put ("user", "sys");
info.put ("password", "change_on_install");
info.put ("internal_logon", "sysdba");

//specify the connection object
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:thin:@database", info);
...

```

Properties for Oracle Performance Extensions Some of these 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 ["Oracle Row Prefetching"](#) on page 12-20.

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

See ["DatabaseMetaData TABLE_REMARKS Reporting"](#) on page 12-26.

- Setting the `defaultBatchValue` property is equivalent to calling `setDefaultExecuteBatch()`.

See ["Oracle Update Batching"](#) on page 12-4.

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.*;
DriverManager.registerDriver (new oracle.jdbc.OracleDriver());

//specify the properties object
java.util.Properties info = new java.util.Properties();
info.put ("user", "scott");
info.put ("password", "tiger");
info.put ("defaultRowProfetch", "20");
info.put ("defaultBatchValue", "5");

//specify the connection object
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:thin:@database",info);
...
```

Opening a Connection for the JDBC OCI Driver

For the JDBC OCI driver, you can specify the database by a `TNSNAMES` entry. You can find the available `TNSNAMES` entries listed in the file `tnsnames.ora` on the client computer from which you are connecting. On Windows NT, this file is located in the `[ORACLE_HOME]\NETWORK\ADMIN` directory. On UNIX systems, you can find it in the `/var/opt/oracle` directory.

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`, enter:

```
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:oci8:@MyHostString", "scott", "tiger");
```

Note that both the ":" and "@" characters are necessary.

For the JDBC OCI and Thin drivers, you can also specify the database with a Oracle Net keyword-value pair. This is less readable than a `TNSNAMES` entry but does not depend on the accuracy of the `TNSNAMES.ORA` file. The Oracle Net keyword-value pair also works with other JDBC drivers.

For example, if you want to connect to the database on host `myhost` that has a TCP/IP listener up on port 1521, and the `SID` (system identifier) is `orcl`, use a statement such as:

```
Connection conn = DriverManager.getConnection
("jdbc:oracle:oci8:@(description=(address=(host= myhost)
(protocol=tcp)(port=1521))(connect_data=(sid=orcl)))", "scott", "tiger");
```

Note: Oracle JDBC does not support login timeouts. Calling the static `DriverManager.setLoginTimeout()` method will have no effect.

Opening a Connection for the JDBC Thin Driver

Because you can use the JDBC Thin driver in applets that do not depend on an Oracle client installation, you cannot use a `TNSNAMES` entry to identify the database to which you want to connect. You have to either:

- Explicitly list the host name, TCP/IP port and Oracle SID of the database to which you want to connect.

or:

- Use a keyword-value pair list.

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

For example, use this string if you want to connect to the database on host `myhost` that has a TCP/IP listener on port 1521 for the database SID (system identifier) `orcl`. You can logon as user `scott`, with password `tiger`:

```
Connection conn = DriverManager.getConnection
("jdbc:oracle:thin:@myhost:1521:orcl", "scott", "tiger");
```

You can also specify the database with a Oracle Net keyword-value pair. This is less readable than the first version, but also works with the other JDBC drivers.

```
Connection conn = DriverManager.getConnection
("jdbc:oracle:thin:@(description=(address=(host=myhost)
(protocol=tcp)(port=1521))(connect_data=(sid=orcl)))", "scott", "tiger");
```

Notes: Oracle JDBC does not support login timeouts. Calling the static `DriverManager.setLoginTimeout()` method will have no effect.

Create a Statement Object

Once you connect to the database and, in the process, create your `Connection` object, the next step is to create a `Statement` object. The `createStatement()` method of your JDBC `Connection` object returns an object of the JDBC `Statement` class. 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();
```

Note that there is nothing Oracle-specific about this statement; it follows standard JDBC syntax.

Execute a Query and Return a Result Set Object

To query the database, use the `executeQuery()` method of your `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 execute a query that populates a `ResultSet` object with the contents of the `ENAME` (employee name) column of a table of employees named `EMP`:

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

Again, there is nothing Oracle-specific about this statement; it follows standard JDBC syntax.

Process the Result Set

Once you execute your query, use the `next()` method of your `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 `getXXX()` methods of the `ResultSet` object, where `XXX` corresponds to a Java datatype.

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));
```

Once again, this is standard JDBC syntax. The `next()` method returns false when it reaches the end of the result set. The employee names are materialized as Java strings.

For a complete sample application showing how to execute a query and print the results, see "[Listing Names from the EMP Table—Employee.java](#)" on page 20-2.

Close the Result Set and Statement Objects

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

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

```
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.

Make Changes to the Database

To write changes to the database, such as for `INSERT` or `UPDATE` operations, you will typically create a `PreparedStatement` object. This allows you to execute a statement with varying sets of input parameters. The `prepareStatement()` method of your JDBC `Connection` object allows you to define a statement that

takes variable bind parameters, and returns a JDBC `PreparedStatement` object with your statement definition.

Use `setXXX()` methods on the `PreparedStatement` object to bind data into the prepared statement to be sent to the database. The various `setXXX()` methods are described in "[Standard setObject\(\) and Oracle setObject\(\) Methods](#)" on page 6-11 and "[Other setXXX\(\) Methods](#)" on page 6-12.

Note that there is nothing Oracle-specific about the functionality described here; it follows standard JDBC syntax.

The following example shows how to use a prepared statement to execute `INSERT` operations that add two rows to the `EMP` table. For the complete sample application, see "[Inserting Names into the EMP Table—InsertExample.java](#)" on page 20-3.

```
// 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();
```

Commit Changes

By default, DML operations (`INSERT`, `UPDATE`, `DELETE`) are committed automatically as soon as they are executed. This is known as *auto-commit* mode. You can, however, disable auto-commit mode with the following method call on the `Connection` object:

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

(For further discussion of auto-commit mode and an example of disabling it, see ["Disabling Auto-Commit Mode"](#) on page 19-6.)

If you disable 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 executed since the last `COMMIT` or `ROLLBACK`.

Important:

- If 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 executed.
 - Any DDL operation, such as `CREATE` or `ALTER`, always includes an implicit `COMMIT`. If auto-commit mode is disabled, this implicit `COMMIT` will not only commit the DDL statement, but also any pending DML operations that had not yet been explicitly committed or rolled back.
-
-

Close the Connection

You must close your connection to the database once you finish your work. Use the `close()` method of the `Connection` object to do this:

```
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 registers an Oracle JDBC Thin driver, connects to the database, creates a `Statement` object, executes a query, and processes the result set.

Note that the code for creating the `Statement` object, executing the query, returning and processing the `ResultSet` object, and closing the statement and connection all follow standard JDBC syntax.

```
import java.sql.*;
import java.math.*;
import java.io.*;
import java.awt.*;

class JdbcTest {
    public static void main (String args []) throws SQLException {
        // Load Oracle driver
        DriverManager.registerDriver (new oracle.jdbc.OracleDriver());
        // Connect to the local database
        Connection conn = DriverManager.getConnection
            ("jdbc:oracle:thin:@myhost:1521:ORCL","scott", "tiger");

        // 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, replace the `Connection` statement with the following:

```
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:oci8:@MyHostString", "scott", "tiger");
```

Where `MyHostString` is an entry in the `TNSNAMES.ORA` file.

Datatype Mappings

The Oracle JDBC drivers support standard JDBC 1.0 and 2.0 types as well as Oracle-specific `BFILE` and `ROWID` datatypes and types of the `REF CURSOR` category.

This section documents standard and Oracle-specific SQL-Java default type mappings.

Table of Mappings

For reference, [Table 3-2](#) shows the default mappings between SQL datatypes, JDBC typecodes, standard Java types, and Oracle extended types.

The **SQL Datatypes** column lists the SQL types that exist in the database.

The **JDBC Typecodes** column lists data typecodes supported by the JDBC standard and defined in the `java.sql.Types` class, or by Oracle in the `oracle.jdbc.OracleTypes` class. For standard typecodes, 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 datatype in the database. These are Oracle extensions that let you retrieve all SQL data in the form of a `oracle.sql.*` Java type. Mapping SQL datatypes into the `oracle.sql` datatypes lets you store and retrieve data without losing information. Refer to "[Package oracle.sql](#)" on page 5-7 for more information on the `oracle.sql.*` package.

Table 3-2 Default Mappings Between SQL Types and Java Types

SQL Datatypes	JDBC Typecodes	Standard Java Types	Oracle Extension Java Types
STANDARD JDBC 1.0 TYPES:			
CHAR	<code>java.sql.Types.CHAR</code>	<code>java.lang.String</code>	<code>oracle.sql.CHAR</code>
VARCHAR2	<code>java.sql.Types.VARCHAR</code>	<code>java.lang.String</code>	<code>oracle.sql.CHAR</code>
LONG	<code>java.sql.Types.LONGVARCHAR</code>	<code>java.lang.String</code>	<code>oracle.sql.CHAR</code>
NUMBER	<code>java.sql.Types.NUMERIC</code>	<code>java.math.BigDecimal</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.DECIMAL</code>	<code>java.math.BigDecimal</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.BIT</code>	<code>boolean</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.TINYINT</code>	<code>byte</code>	<code>oracle.sql.NUMBER</code>

Table 3–2 Default Mappings Between SQL Types and Java Types (Cont.)

SQL Datatypes	JDBC Typecodes	Standard Java Types	Oracle Extension Java Types
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
DATE	java.sql.Types.TIMESTAMP	java.sql.Timestamp	oracle.sql.DATE
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	n/a	oracle.sql.BFILE
ROWID	oracle.jdbc.OracleTypes.ROWID	n/a	oracle.sql.ROWID
REF CURSOR type	oracle.jdbc.OracleTypes.CURSOR	java.sql.ResultSet	oracle.jdbc.OracleResultSet

Note: Under JDK 1.1.x, the Oracle package `oracle.jdbc2` is required to support JDBC 2.0 types. (Under JDK 1.2.x they are supported by the standard `java.sql` package.)

For a list of all the Java datatypes to which you can validly map a SQL datatype, see ["Valid SQL-JDBC Datatype Mappings"](#) on page 21-2.

See [Chapter 5, "Overview of Oracle Extensions"](#), for more information on type mappings. In [Chapter 5](#) you can also find more information on the following:

- packages `oracle.sql`, `oracle.jdbc`, and `oracle.jdbc2`
- type extensions for the Oracle `BFILE` and `ROWID` datatypes and user-defined types of the `REF CURSOR` category

Notes Regarding Mappings

This section goes into further detail regarding mappings for `NUMBER` and user-defined types.

Regarding 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` (for user-defined objects only)
- The Oracle-specific `oracle.sql.ORAData` (primarily for user-defined objects, object references, and collections, but able to map from *any* SQL type where you want customized processing of any kind)

For information about custom Java classes and the `SQLData` and `ORAData` interfaces, see ["Mapping Oracle Objects"](#) on page 8-2 and ["Creating and Using Custom Object Classes for Oracle Objects"](#) on page 8-10. (Although these sections focus on custom Java classes for user-defined objects, there is some general information about other kinds of custom Java classes as well.)

Regarding NUMBER Types

For the different typecodes 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 $-128 < x < 128$.

Java Streams in JDBC

This section covers the following topics:

- [Streaming LONG or LONG RAW Columns](#)
- [Streaming CHAR, VARCHAR, or RAW Columns](#)
- [Data Streaming and Multiple Columns](#)
- [Streaming and Row Prefetching](#)
- [Closing a Stream](#)
- [Streaming LOBs and External Files](#)

This section describes how the Oracle JDBC drivers handle Java streams for several datatypes. Data streams allow you to read `LONG` column data of up to 2 gigabytes. Methods associated with streams let you read the data incrementally.

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 stream**—Used for `RAW` bytes of data. This corresponds to the `getBinaryStream()` method.
- **ASCII stream**—Used for ASCII bytes in ISO-Latin-1 encoding. This corresponds to the `getAsciiStream()` method.
- **Unicode stream**—Used for Unicode bytes with the `UCS-2` encoding. This corresponds to the `getUnicodeStream()` method.

The methods `getBinaryStream()`, `getAsciiStream()`, and `getUnicodeStream()` return the bytes of data in an `InputStream` object. These methods are described in greater detail in [Chapter 7, "Working with LOBs and BFILES"](#).

For a complete sample application showing how to read and write stream data, see ["Streams—StreamExample.java"](#) on page 20-18.

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.

To access the data in a LONG column, you can get the column as a Java `InputStream` and use the `read()` method of the `InputStream` object. As an alternative, you can get the data as a string or byte array, in which 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 NLS conversions for you, depending on the character set of your database and the driver. For more information about NLS, see "[JDBC and Globalization Support](#)" on page 18-2.

LONG RAW Data Conversions

A call to `getBinaryStream()` returns RAW data "as-is". 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 bytes.

For example, if your LONG RAW column contains the bytes 20 21 22, you receive the following bytes:

LONG RAW	BinaryStream	AsciiStream	UnicodeStream
20 21 22	20 21 22	49 52 49 53 49 54	0049 0052 0049 0053 0049 0054
		which is also	which is also:
		'1' '4' '1' '5' '1' '6'	'1' '4' '1' '5' '1' '6'

For example, the LONG RAW value 20 is represented in hexadecimal as 14 or "1" "4". In ASCII, 1 is represented by "49" and "4" is represented by "52". In Unicode, a padding of zeros is used to separate individual values. So, the hexadecimal value 14 is represented as 0 "1" 0 "4". The Unicode representation is 0 "49" 0 "52".

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, 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 UCS-2 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.

For more information on how the drivers return data based on character set, see "[JDBC and Globalization Support](#)" on page 18-2.

Note: Receiving LONG or LONG RAW columns as a stream (the default case) requires you to pay special attention to the order in which you receive data from the database. For more information, see "[Data Streaming and Multiple Columns](#)" on page 3-25.

[Table 3-3](#) summarizes LONG and LONG RAW data conversions for each stream type.

Table 3-3 LONG and LONG RAW Data Conversions

Datatype	BinaryStream	AsciiStream	UnicodeStream
LONG	bytes representing characters in Unicode UTF-8. The bytes can represent characters in US7ASCII or WE8ISO8859P1 if: <ul style="list-style-type: none"> ■ the value of NLS_LANG on the client is US7ASCII or WE8ISO8859P1. or: <ul style="list-style-type: none"> ■ the database character set is US7ASCII or WE8ISO8859P1. 	bytes representing characters in ISO-Latin-1 (WE8ISO8859P1) encoding	bytes representing characters in Unicode UCS-2 encoding
LONG RAW	as-is	ASCII representation of hexadecimal bytes	Unicode representation of hexadecimal bytes

Streaming Example for LONG RAW Data

One of the features of a `getXXXStream()` method is that it allows 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; the second version uses the `getBytes()` method.

Getting a LONG RAW Data Column with `getBinaryStream()` This Java example writes the contents of a LONG RAW column to a file on the local file system. In this case, the driver fetches the data incrementally.

The following code creates the table that stores a column of LONG RAW data associated with the name LESLIE:

```
-- SQL code:
create table streamexample (NAME varchar2 (256), GIFDATA long raw);
insert into streamexample values ('LESLIE', '00010203040506070809');
```

The following Java code snippet writes the data from the LESLIE 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 contents of the GIFDATA column are transferred incrementally in chunk-sized pieces between the database and the client. 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 version of the 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 previous code snippet can be rewritten as:

```
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 will probably 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, you might 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 type `VARCHAR` or `VARBINARY`, then the driver will not automatically stream the data.

If you redefine column types with `defineColumnType()`, you must declare the types of *all* columns in the query. If you do not, `executeQuery()` will fail. In addition, you must cast the `Statement` object to an `oracle.jdbc.OracleStatement` object.

As an added benefit, using `defineColumnType()` saves the driver two round trips to the database when executing the query. Without `defineColumnType()`, the JDBC driver has to request the datatypes of the column types.

Using the example from the previous section, the `Statement` object `stmt` is cast to the `OracleStatement` and the column containing `LONG RAW` data is redefined to be of the type `VARBINARY`. The data is not streamed—instead, it is returned in a byte array.

```
//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, the JDBC driver will return a Java `InputStream`, but no real streaming occurs. In the case of these datatypes, the JDBC driver fully fetches the data into an in-memory buffer during a call to the `executeQuery()` method or `next()` method. The `getXXXStream()` entry points return a stream that reads data from this buffer.

Data Streaming and Multiple Columns

If your query selects 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 ["Streaming Data Precautions"](#) on page 3-28 for more information.

Streaming Example with Multiple Columns

Consider the following query:

```
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 iterator, you must complete any processing of the stream column before reading the number column.

An exception to this behavior is LOB data, which is also transferred between server and client as a Java stream. For more information on how the driver treats LOB data, see ["Streaming LOBs and External Files"](#) on page 3-27.

Bypassing Streaming Data Columns

There might be situations where you want to avoid reading a column that contains streaming data. If you do not want to read the data for the streaming column, then call the `close()` method of the stream object. This method discards the stream data and allows the driver to continue reading data for all the non-streaming columns that follow the stream. Even though you are intentionally discarding the stream, it is good programming practice to call the columns in SELECT-list order.

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);
}
```

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 and points to the location of the actual data. External files (binary files, or BFILEs) are managed similarly. The JDBC drivers can support these types through the use of streams:

- BLOBs (unstructured binary data)
- CLOBs (character data)
- BFILEs (external files)

LOBs and BFILEs behave differently from the other types of streaming data described in this chapter. The driver transfers data between server and client as a Java stream. However, unlike most Java streams, a locator representing the data is stored in the table. Thus, you can access the data at any time during the life of the connection.

Streaming BLOBs and CLOBs

When a query selects one or more CLOB or BLOB columns, the JDBC driver transfers to the client the data pointed to by the locator. The driver performs the transfer as a Java 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 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.

For a complete discussion of how to use streaming CLOB and BLOB data, see "[Reading and Writing BLOB and CLOB Data](#)" on page 7-6.

Important: The JDBC 2.0 specification states that `PreparedStatement` methods `setBinaryStream()` and `setObject()` can be used to input a stream value as a BLOB, and that the `PreparedStatement` methods `setAsciiStream()`, `setUnicodeStream()`, `setCharacterStream()`, and `setObject()` can be used to input a stream value as a CLOB. This bypasses the LOB locator, going directly to the LOB data itself.

In the implementation of the Oracle JDBC drivers, this functionality is supported *only* for a configuration using an 8.1.6 database and 8.1.6 JDBC OCI driver. **Do not use this functionality for any other configuration, as data corruption can result.**

Streaming BFILEs

An external file, or BFILE, is used to store a locator to a file outside the database, stored somewhere on the filesystem of the data server. The locator points to the actual location of the file.

When a query selects one or more BFILE columns, the JDBC driver transfers to the client the file pointed to by the locator. The transfer is performed in a Java stream. To manipulate BFILE data from JDBC, use methods in the Oracle extension class `oracle.sql.BFILE`. This class provides 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.

For a complete discussion of how to use streaming BFILE data, see ["Reading BFILE Data"](#) on page 7-22.

Closing a Stream

You can discard the data from a stream at any time by calling the stream's `close()` method. You can also close and discard the stream by closing its result set or connection object. You can find more information about the `close()` method for data streams in ["Bypassing Streaming Data Columns"](#) on page 3-26. For information on how to avoid closing a stream and discarding its data by accident, see ["Streaming Data Precautions"](#) on page 3-28.

Notes and Precautions on Streams

This section discusses several noteworthy and 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 described:

- Use the stream data after you access it.

To recover the data from a column containing a data stream, it is not enough to get the column; you must immediately process its contents. Otherwise, the contents will be discarded when you get the next column.

- Call the stream column in `SELECT`-list order.

If your query selects 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, the database sends the entire data stream before proceeding to the next column.

If you do not use the `SELECT`-list order 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, 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, 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, the data will not be available and the driver will return a "Stream Closed" error.

The second 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, 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()`

There is a limit on the maximum size of the array which can be bound using the `PreparedStatement` class `setBytes()` method, and on the size of the string which can be bound using the `setString()` method.

Above the limits, which depend on the version of the server you use, you should use `setBinaryStream()` or `setCharacterStream()` instead.

When connecting to an Oracle8 database, the limit for `setBytes()` is 2000 bytes (the maximum size of a RAW in Oracle8) and the limit for `setString()` is 4000 bytes (the maximum size of a VARCHAR2 in Oracle8).

When connecting to an Oracle7 database, the limit for `setBytes()` is 255 bytes (the maximum size of a RAW in Oracle7) and the limit for `setString()` is 2000 bytes (the maximum size of a VARCHAR2 in Oracle7).

The 8.1.6 Oracle JDBC drivers may not raise an error if you exceed the limit when using `setBytes()` or `setString()`, but you may receive the following error:

```
ORA-17070: Data size bigger than max size for this type
```

Future versions of the Oracle drivers will raise an error if the length exceeds these limits.

Note: This discussion applies to binds in SQL, not PL/SQL.

Streaming and Row Prefetching

If the JDBC driver encounters a column containing a data stream, row prefetching is set back to 1.

Row prefetching is an Oracle performance enhancement that allows multiple rows of data to be retrieved with each trip to the database. See "[Oracle Row Prefetching](#)" on page 12-20.

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 execution of PL/SQL stored procedures and anonymous blocks. They support both SQL92 escape syntax and Oracle PL/SQL block 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 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 (vall char)
return char as
begin
    return vall || 'suffix';
end;
```

Your invocation call in your JDBC program should look like:

```
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:oci8:@<hoststring>", "scott", "tiger");

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

```
String result = cs.getString(1);
```

For complete sample applications that call PL/SQL stored procedures and functions in SQL92 syntax and Oracle PL/SQL block syntax, see "[Calling PL/SQL Stored Procedures—PLSQLExample.java](#)" on page 20-6 and "[Executing Procedures in PL/SQL Blocks—PLSQL.java](#)" on page 20-7.

Java Stored Procedures

You can use JDBC to invoke 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, have had call specifications written to publish them to the Oracle data dictionary). See the *Oracle9i Java Stored Procedures Developer's Guide* for more information on writing, publishing, and using Java stored procedures.

Processing SQL Exceptions

To handle error conditions, the Oracle JDBC drivers throws SQL exceptions, producing instances of class `java.sql.SQLException` or a subclass. Errors can originate either in the JDBC driver or in the database (RDBMS) 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.

Errors originating in the JDBC driver are listed with their ORA numbers in [Appendix B, "JDBC Error Messages"](#).

Errors originating in the RDBMS are documented in the *Oracle9i Database Error Messages* reference.

Retrieving Error Information

You can retrieve basic error information with these `SQLException` methods:

- `getMessage()`
For errors originating in the JDBC driver, this method returns the error message with no prefix. For errors originating in the RDBMS, it returns the error message prefixed with the corresponding ORA number.
- `getErrorCode()`
For errors originating in either the JDBC driver or the RDBMS, this method returns the five-digit ORA number.
- `getSQLState()`
For errors originating in the JDBC driver, this returns no useful information. For errors originating in the RDBMS, this method returns a five-digit code indicating the SQL state. Your code should be prepared to handle null data.

The following example prints output from a `getMessage()` call.

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

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

```
exception: Invalid column type
```

(There is no ORA number message prefix for errors originating in the JDBC driver, although you can get the ORA number with a `getErrorCode()` call.)

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

Printing the Stack Trace

The `SQLException` class provides the following method for printing a stack trace.

- `printStackTrace()`

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, executing the program would produce the following error text:

```
java.sql.SQLException: Invalid column index
at oracle.jdbc.dbaccess.DBError.check_error(DBError.java:235)
at oracle.jdbc.OracleStatement.prepare_for_new_get(OracleStatemen
```

```
t.java:1560)
at oracle.jdbc.OracleStatement.getStringValue(OracleStatement.java:1653)
at oracle.jdbc.OracleResultSet.getString(OracleResultSet.java:175)
)
at Employee.main(Employee.java:41)
```

Overview of JDBC 2.0 Support

A key aspect of Oracle JDBC is JDBC 2.0 functionality, both new functionality that was not previously supported, and the standardization of functionality that was previously supported through Oracle extensions.

This chapter provides an overview of JDBC 2.0 support in the Oracle JDBC drivers, focusing in particular on any differences in support between the JDK 1.2.x and JDK 1.1.x environments. The following topics are discussed:

- [Introduction](#)
- [JDBC 2.0 Support: JDK 1.2.x versus JDK 1.1.x](#)
- [Overview of JDBC 2.0 Features](#)

Introduction

The Oracle JDBC drivers are compliant with the JDBC 2.0 specification. JDBC 2.0 functionality previously implemented through Oracle extensions in the `oracle.jdbc2` package—such as structured objects, object references, arrays, and LOBs—is now implemented through the standard `java.sql` package in JDK 1.2.

In a JDK 1.1.x environment, you can continue to use the `oracle.jdbc2` package. You can also use JDBC 2.0 features in connection objects, statement objects, result set objects, and database meta data objects under JDK 1.1.x by casting your objects to the Oracle types.

Furthermore, you can use features of the JDBC 2.0 Optional Package (also known as the JDBC 2.0 Standard Extension API) under either JDK 1.2.x or JDK 1.1.x. These features, including connection pooling and distributed transactions, are supported through the standard `javax.sql` package. This package and the classes that implement its interfaces are now included with the JDBC classes ZIP file for either JDK 1.2.x or JDK 1.1.x.

JDBC 2.0 Support: JDK 1.2.x versus JDK 1.1.x

Support for standard JDBC 2.0 features differs depending on whether you are using JDK 1.2.x or JDK 1.1.x. There are three areas to consider:

- datatype support—such as for objects, arrays, and LOBs—which is handled through the standard `java.sql` package under JDK 1.2.x and through the Oracle extension `oracle.jdbc2` package under JDK 1.1.x
- standard feature support—such as result set enhancements and update batching—which is handled through standard objects such as `Connection`, `ResultSet`, and `PreparedStatement` under JDK 1.2.x, but requires Oracle-specific functionality under JDK 1.1.x
- extended feature support—features of the JDBC 2.0 Optional Package (also known as the Standard Extension API), including data sources, connection pooling, and distributed transactions—which has the same support and functionality in either JDK 1.2.x or JDK 1.1.x

This section also discusses performance enhancements available under JDBC 2.0—update batching and fetch size—that are also still available as Oracle extensions, then concludes with a brief discussion about migration from JDK 1.1.x to JDK 1.2.x.

Datatype 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.

For JDBC 2.0 functionality under JDK 1.2.x, where you are using `classes12.zip`, no special imports are required. The following imports, both of which you will likely need even if you are not using JDBC 2.0 features, will suffice:

```
import java.sql.*;
import oracle.sql.*;
```

JDBC 2.0 features are not supported by JDK 1.1.x; however, Oracle provides extensions that allow you to use a significant subset of JDBC 2.0 datatypes under JDK 1.1.x, where you are using `classes11.zip`. These extensions support database objects, object references, arrays, and LOBs.

The package `oracle.jdbc2` is included in `classes11.zip`. This package provides interfaces that mimic JDBC 2.0-related interfaces that became standard

with JDK 1.2.x for SQL3 and advanced datatypes. The interfaces in `oracle.jdbc2` are implemented as appropriate by classes in the `oracle.sql` package for a JDK 1.1.x environment.

The following imports are required for JDBC 2.0 datatypes under JDK 1.1.x:

```
import java.sql.*;
import oracle.jdbc2.*;
import oracle.sql.*;
```

Standard Feature Support

In a JDK 1.2.x environment (using the JDBC classes in `classes12.zip`), 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. Therefore, under JDK 1.2.x, you can use standard objects such as `Connection`, `DatabaseMetaData`, `ResultSetMetaData`, `Statement`, `PreparedStatement`, `CallableStatement`, and `ResultSet` to use these features.

In a JDK 1.1.x environment (using the JDBC classes in `classes111.zip`), Oracle JDBC provides support for these JDBC 2.0 features as Oracle extensions. To use this functionality, you must cast your objects to the Oracle types:

- `OracleConnection`
- `OracleDatabaseMetaData`
- `OracleResultSetMetaData`
- `OracleStatement`
- `OraclePreparedStatement`
- `OracleCallableStatement`
- `OracleResultSet`

For example, to use JDBC 2.0 result set enhancements, you must do the following:

- Explicitly type or cast scrollable or updatable result sets as type `OracleResultSet`.
- Explicitly type or cast connection objects as type `OracleConnection` whenever the connection object will be required to produce a statement object that will in turn produce a scrollable or updatable result set.

In addition, you might have to cast statement objects to `OracleStatement`, `OraclePreparedStatement`, or `OracleCallableStatement`, and cast

database meta data objects to `OracleDatabaseMetaData`. This would be if you want to use JDBC 2.0 statement or database meta data methods described under ["Summary of New Methods for Result Set Enhancements"](#) on page 11-32.

Extended Feature Support

Features of the JDBC 2.0 Optional Package (also known as the Standard Extension API), including data sources, connection pooling, and distributed transactions, are supported equally in a JDK 1.2.x or 1.1.x environment.

The standard `javax.sql` package and classes that implement its interfaces are included in the JDBC classes ZIP file for either environment.

Standard versus Oracle Performance Enhancement APIs

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

- update batching
- fetch size / 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.

For more information, see the following sections:

- ["Update Batching"](#) on page 12-2
- ["Fetch Size"](#) on page 11-24
- ["Oracle Row Prefetching"](#) on page 12-20

Migration from JDK 1.1.x to JDK 1.2.x

The only migration requirements in going from JDK 1.1.x to JDK 1.2.x are as follows:

- Remove your imports of the `oracle.jdbc2` package, as discussed above under ["Datatype Support"](#) on page 4-3.
- Replace any direct references to `oracle.jdbc2.*` interfaces with references to the standard `java.sql.*` interfaces.
- Type map objects (for mapping SQL structured objects to Java types), which must extend the `java.util.Dictionary` class under JDK 1.1.x, must

implement the `java.util.Map` interface under JDK 1.2.x. Note, however, that the class `java.util.Hashtable` satisfies either requirement. If you used `Hashtable` objects for your type maps under JDK 1.1.x, then no change is necessary. For more information, see ["Creating a Type Map Object and Defining Mappings for a `SQLData` Implementation"](#) on page 8-12.

If these points do not apply to your code, then you do not need to make any code changes or recompile to run under JDK 1.2.x.

Overview of JDBC 2.0 Features

Table 4–1 lists key areas of JDBC 2.0 functionality and points to where you can go in this manual for more information about Oracle support.

Table 4–1 Key Areas of JDBC 2.0 Functionality

Feature	Comments and References
update batching	Also available previously as an Oracle extension. Under either JDK 1.2.x or JDK 1.1.x you can use either the standard update batching model or the Oracle model. See "Update Batching" on page 12-2 for information.
result set enhancements (scrollable and updatable result sets)	This is also available under JDK 1.1.x as an Oracle extension. See Chapter 11, "Result Set Enhancements" for information.
fetch size / row prefetching	The JDBC 2.0 fetch size feature is also available under JDK 1.1.x as an Oracle extension. Under either JDK 1.2.x or JDK 1.1.x, you can also use Oracle row prefetching, which is largely equivalent to the JDBC 2.0 fetch size feature but predates JDBC 2.0. See "Fetch Size" on page 11-24 and "Oracle Row Prefetching" on page 12-20 for information.
use of JNDI (Java Naming and Directory Interface) to specify and obtain database connections	This requires data sources, which are part of the JDBC 2.0 Optional Package (JDBC 2.0 Standard Extension API) in the <code>javax.sql</code> package. This is available under either JDK 1.2.x or JDK 1.1.x. See "A Brief Overview of Oracle Data Source Support for JNDI" on page 14-2 and "Creating a Data Source Instance, Registering with JNDI, and Connecting" on page 14-8 for information.
connection pooling (framework for connection caching)	This requires the JDBC 2.0 Optional Package (JDBC 2.0 Standard Extension API) in the <code>javax.sql</code> package. This is available under either JDK 1.2.x or 1.1.x. See "Connection Pooling" on page 14-13 for information.
connection caching (sample Oracle implementation)	This requires the JDBC 2.0 Optional Package (JDBC 2.0 Standard Extension API) in the <code>javax.sql</code> package. This is available under either JDK 1.2.x or 1.1.x. See "Connection Caching" on page 14-18 for information.

Table 4–1 Key Areas of JDBC 2.0 Functionality (Cont.)

Feature	Comments and References
distributed transactions / XA functionality	This requires the JDBC 2.0 Optional Package (JDBC 2.0 Standard Extension API) in the <code>javax.sql</code> package. This is available under either JDK 1.2.x or 1.1.x. See Chapter 15, "Distributed Transactions" for information.
miscellaneous <code>getXXX()</code> methods	See "Other getXXX() Methods" on page 6-7 for information about which <code>getXXX()</code> methods are Oracle extensions under JDK 1.2.x and 1.1.x, and about any differences in functionality with JDBC 2.0.
miscellaneous <code>setXXX()</code> methods	See "Other setXXX() Methods" on page 6-12 for information about which <code>setXXX()</code> methods are Oracle extensions under JDK 1.2.x and 1.1.x, and about any differences in functionality with JDBC 2.0.

Note: The Oracle JDBC drivers do not support the `Calendar` datatype because it is not yet feasible to support `java.sql.Date` timezone information. `Calendar` input to `setXXX()` or `getXXX()` method calls for `Date`, `Time`, and `Timestamp` is ignored. The `Calendar` type will be supported in a future Oracle release.

Overview of Oracle Extensions

Oracle's extensions to the JDBC standard include Java packages and interfaces that let you access and manipulate Oracle datatypes and use Oracle performance extensions. Compared to standard JDBC, the extensions offer you greater flexibility in how you can manipulate the data. This chapter presents an overview of the packages and classes included in Oracle's extensions to standard JDBC. It also describes some of the key support features of the extensions.

This chapter includes these topics:

- [Introduction to Oracle Extensions](#)
- [Support Features of the Oracle Extensions](#)
- [Oracle JDBC Packages and Classes](#)
- [Oracle Character Datatypes Support](#)
- [Additional Oracle Type Extensions](#)

Note: This chapter focuses on type extensions, as opposed to performance extensions, which are discussed in detail in [Chapter 12, "Performance Extensions"](#).

Introduction to Oracle Extensions

Oracle provides two implementations of its JDBC drivers—one that supports Sun Microsystems JDK 1.2.x and complies with the Sun JDBC 2.0 standard, and one that supports JDK 1.1.x and complies with the Sun JDBC 1.22 standard.

Beyond standard features, Oracle JDBC drivers provide Oracle-specific type extensions and performance extensions.

Both implementations include the following Java packages:

- `oracle.sql` (classes to support all Oracle type extensions)
- `oracle.jdbc` (interfaces to support database access and updates in Oracle type formats)

In addition to these packages, the implementation for JDK 1.1.x includes the following Java package. This package supports some JDBC 2.0 features by providing interfaces that mimic the JDBC 2.0 interfaces in the standard `java.sql` package:

- `oracle.jdbc2` (interfaces equivalent to standard JDBC 2.0 interfaces)

(For example, `oracle.jdbc2.Struct` mimics `java.sql.Struct`, which exists in JDK 1.2.)

["Oracle JDBC Packages and Classes"](#) on page 5-7 further describes the preceding packages and their classes.

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 datatypes, Oracle objects, and specific schema naming.

Support for Oracle Datatypes

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 datatypes, acting as wrappers for raw SQL data. This functionality provides two significant advantages in manipulating SQL data:

- Accessing data directly in SQL format is 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 occurs in converting between SQL and Java formats.

Once manipulations are complete and it is time to output the information, each of the `oracle.sql.*` type support classes has all the necessary methods to convert data to appropriate Java formats. For a more detailed description of these general issues, see "[Package `oracle.sql`](#)" on page 5-7.

See the following for more information on specific `oracle.sql.*` datatype classes:

- "[Oracle Character Datatypes Support](#)" on page 5-28 for information on `oracle.sql.*` character datatypes which includes the SQL CHAR and SQL NCHAR datatypes
- "[Additional Oracle Type Extensions](#)" on page 5-33 for information on the `oracle.sql.*` datatype classes for ROWIDs and REF CURSOR types
- [Chapter 7, "Working with LOBs and BFILES"](#) for information on `oracle.sql.*` datatype support for BLOBs, CLOBs, and BFILES
- [Chapter 8, "Working with Oracle Object Types"](#) for information on `oracle.sql.*` datatype support for composite data structures (Oracle objects) in the database
- [Chapter 9, "Working with Oracle Object References"](#) for information on `oracle.sql.*` datatype support for object references

- [Chapter 10, "Working with Oracle Collections"](#) for information on `oracle.sql.*` datatype support for collections (VARRAYs and nested tables)

Support for Oracle Objects

Oracle JDBC supports the use of structured objects in the database, where an object datatype 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 (a character string), a `lastname` attribute (another character string), and an `employeenumber` attribute (integer).

Oracle's JDBC implementation supports Oracle object datatypes. When you work with Oracle object datatypes in a Java application, you must consider the following:

- how to map between Oracle object datatypes and Java classes
- how to store Oracle object attributes in corresponding Java objects (they can be stored in standard Java types or in `oracle.sql.*` types)
- 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.ORADData` interface. ([Chapter 8, "Working with Oracle Object Types"](#) provides more detail regarding these interfaces.) Each interface specifies methods to convert data between SQL and Java.

Note: The `ORADData` interface has replaced the `CustomDatum` interface. While the latter interface is deprecated for Oracle9i, it is still supported for backward compatibility.

To create custom Java classes to correspond to your Oracle objects, Oracle recommends that you use the Oracle9i `JPublisher` utility to create the classes. To do this, you must define attributes according to how you want to store the data. `JPublisher` performs this task seamlessly with command-line options and can generate either `SQLData` or `ORADData` implementations.

For `SQLData` implementations, a *type map* defines the correspondence between Oracle object datatypes and Java classes. Type maps are objects of a special Java class that specify which Java class corresponds to each Oracle object datatype.

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 `getXXX()` methods of the custom Java classes, which retrieve data into your Java application. For more information on the `JPublisher` utility, see the *Oracle9i JPublisher User's Guide*.

[Chapter 8, "Working with Oracle Object Types"](#) describes Oracle JDBC support for Oracle objects.

Support for Schema Naming

Oracle JDBC 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. Notice that *schema_name* and *sql_type_name* are separated by a dot (".").

To specify an object type in JDBC, you use its fully qualified name (that is, a schema name and SQL type 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 dot in it, such as `CORPORATE.EMPLOYEE`, the type name must be quoted.
- The JDBC driver looks for the first unquoted dot in the object's name and uses the string before the dot as the schema name and the string following the dot as the type name. If no dot is found, 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 might 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 dot 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, 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.

OCI Extensions

See [Chapter 16, "JDBC OCI Extensions"](#) for the following OCI driver-specific information:

- [OCI Driver Connection Pooling](#)
- [Middle-Tier Authentication Through Proxy Connections](#)
- [OCI Driver Transparent Application Failover](#)
- [OCI HeteroRM XA](#)
- [Accessing PL/SQL Index-by Tables](#)

Oracle JDBC Packages and Classes

This section describes the Java packages that support the Oracle JDBC extensions and the key classes that are included in these packages:

- [Package oracle.sql](#)
- [Package oracle.jdbc](#)
- [Package oracle.jdbc2 \(for JDK 1.1.x only\)](#)

You can refer to the Oracle JDBC Javadoc for more information about all the classes mentioned in this section.

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 datatypes.

Essentially, the classes act as Java wrappers for the raw SQL data. Because data in an `oracle.sql.*` object remains in SQL format, no information is lost. For SQL primitive types, these classes simply wrap the SQL data. For SQL structured types (objects and arrays), they provide additional information such as conversion methods and details of structure.

Each of the `oracle.sql.*` datatype classes extends `oracle.sql.Datum`, a superclass that encapsulates functionality common to all the datatypes. Some of the classes are for JDBC 2.0-compliant datatypes. These classes, as [Table 5–1](#) indicates, implement standard JDBC 2.0 interfaces in the `java.sql` package (`oracle.jdbc2` for JDK 1.1.x), as well as extending the `oracle.sql.Datum` class.

Classes of the oracle.sql Package

[Table 5–1](#) lists the `oracle.sql` datatype classes and their corresponding Oracle SQL types.

Table 5–1 Oracle Datatype Classes

Java Class	Oracle SQL Types and Interfaces Implemented
<code>oracle.sql.STRUCT</code>	STRUCT (objects) implements <code>java.sql.Struct</code> (<code>oracle.jdbc2.Struct</code> under JDK 1.1.x)
<code>oracle.sql.REF</code>	REF (object references) implements <code>java.sql.Ref</code> (<code>oracle.jdbc2.Ref</code> under JDK 1.1.x)

Table 5–1 Oracle Datatype Classes (Cont.)

Java Class	Oracle SQL Types and Interfaces Implemented
<code>oracle.sql.ARRAY</code>	VARRAY or nested table (collections) implements <code>java.sql.Array</code> (<code>oracle.jdbc2.Array</code> under JDK 1.1.x)
<code>oracle.sql.BLOB</code>	BLOB (binary large objects) implements <code>java.sql.Blob</code> (<code>oracle.jdbc2.Blob</code> under JDK 1.1.x)
<code>oracle.sql.CLOB</code>	SQL CLOB (character large objects) and Globalization Support NCLOB datatypes both implement <code>java.sql.Clob</code> (<code>oracle.jdbc2.Clob</code> under JDK 1.1.x)
<code>oracle.sql.BFILE</code>	BFILE (external files)
<code>oracle.sql.CHAR</code>	CHAR, NCHAR, VARCHAR2, NVARCHAR2
<code>oracle.sql.DATE</code>	DATE
<code>oracle.sql.TIMESTAMP</code>	TIMESTAMP
<code>oracle.sql.TIMESTAMPTZ</code>	TIMESTAMPTZ (Timestamp with Time Zone)
<code>oracle.sql.TIMESTAMPLTZ</code>	TIMESTAMPLTZ (Timestamp with Local Time Zone)
<code>oracle.sql.NUMBER</code>	NUMBER
<code>oracle.sql.RAW</code>	RAW
<code>oracle.sql.ROWID</code>	ROWID (row identifiers)
<code>oracle.sql.OPAQUE</code>	OPAQUE

You can find more detailed information about each of these classes later in this chapter. Additional details about use of the Oracle extended types (STRUCT, REF, ARRAY, BLOB, CLOB, BFILE, and ROWID) are described in the following locations:

- ["Oracle Character Datatypes Support"](#) on page 5-28
- ["Additional Oracle Type Extensions"](#) on page 5-33
- [Chapter 7, "Working with LOBs and BFILES"](#)
- [Chapter 8, "Working with Oracle Object Types"](#)
- [Chapter 9, "Working with Oracle Object References"](#)
- [Chapter 10, "Working with Oracle Collections"](#)

Notes:

- For information about retrieving data from a result set or callable statement object into `oracle.sql.*` types, as opposed to Java types, see [Chapter 6, "Accessing and Manipulating Oracle Data"](#).
 - 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 datatype classes, the `oracle.sql` package includes the following support classes and interfaces, primarily for use with objects and collections:

- `oracle.sql.ArrayDescriptor` class: Used in constructing `oracle.sql.ARRAY` objects; describes the SQL type of the array. (See ["Creating ARRAY Objects and Descriptors"](#) on page 10-11.)
- `oracle.sql.StructDescriptor` class: Used in constructing `oracle.sql.STRUCT` objects, which you can use as a default mapping to Oracle objects in the database. (See ["Creating STRUCT Objects and Descriptors"](#) on page 8-4.)
- `oracle.sql.ORAData` and `oracle.sql.ORADataFactory` interfaces: Used in Java classes implementing the Oracle `ORAData` scenario of Oracle object support. (The other possible scenario is the JDBC-standard `SQLData` implementation.) See ["Understanding the ORAData Interface"](#) on page 8-21 for more information on `ORAData`.
- `oracle.sql.OpaqueDescriptor` class: Used to obtain the meta data for an instance of the `oracle.sql.OPAQUE` class.

General oracle.sql.* Datatype Support

Each of the Oracle datatype 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 (in the raw format in which JDBC received the data from the database)
- 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 datatypes 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 `xxxValue()` methods to convert SQL data to Java typed—for example: `stringValue()`, `intValue()`, `booleanValue()`, `dateValue()`, `bigDecimalValue()`
- additional conversion, `getXXX()` and `setXXX()` methods as appropriate for the functionality of the datatype (such as methods in the 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)

Refer to the Oracle JDBC Javadoc for additional information about these classes. See ["Class oracle.sql.CHAR"](#) on page 5-29 to learn how the `oracle.sql.CHAR` class supports character data.

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. (If you use a `SQLData` custom Java class, 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 (`oracle.jdbc2.Struct` under JDK 1.1.x) and extends the `oracle.sql.Datum` class.

In the database, Oracle stores the raw bytes of object data in a linearized form. A `STRUCT` object is a wrapper for the raw bytes of an Oracle object. It contains the SQL type name of the Oracle object and a "values" array of `oracle.sql.Datum` objects that hold the attribute values in SQL format.

You can materialize a `STRUCT`'s attributes 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 all the 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 values 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 the `oracle.sql.CHAR` type.
 - Nested objects in the values array of a `STRUCT` object are materialized by the JDBC driver as instances of `STRUCT`.
-
-

In some cases, you might 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.

For more information about working with Oracle objects using the `oracle.sql.STRUCT` and `StructDescriptor` classes, see ["Using the Default STRUCT Class for Oracle Objects"](#) on page 8-3.

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.*` datatype classes, is a subclass of the `oracle.sql.Datum` class. It implements the standard JDBC 2.0 `java.sql.Ref` interface (`oracle.jdbc2.Ref` under JDK 1.1.x).

The `REF` class has methods to retrieve and pass object references. Be aware, however, that 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.

For more information about working with Oracle object references using the `oracle.sql.REF` class, see [Chapter 9, "Working with 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 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 (`oracle.jdbc2.Array` under JDK 1.1.x).

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.

For more information about working with Oracle collections using the `oracle.sql.ARRAY` and `ArrayDescriptor` classes, see ["Overview of Collection \(Array\) Functionality"](#) on page 10-5.

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

`BLOBs` and `CLOBs` (referred to collectively as "LOBs"), and `BFILEs` (for external files) 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 datatypes in several ways:

- `BLOBs` point to large unstructured binary data items and are supported by the `oracle.sql.BLOB` class.
- `CLOBs` point to large fixed-width character data items (that is, characters that require a fixed number of bytes per character) 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, but bear in mind that you are receiving only the locator, not the data itself. Additional steps are necessary to retrieve the data.

For information about how to access and manipulate locators and data for LOBs and BFILES, see [Chapter 7, "Working with LOBs and BFILES"](#).

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

These classes map to primitive SQL datatypes, which are a part of standard JDBC, and supply conversions to and from the corresponding JDBC Java types. For more information, see the Javadoc.

Classes `oracle.sql.TIMESTAMP`, `oracle.sql.TIMESTAMPTZ`, and `oracle.sql.TIMESTAMPLTZ`

The Oracle9i JDBC drivers support the following date/time datatypes:

- Timestamp (TS)
- Timestamp with Time Zone (TSTZ)
- `TIMESTAMP` with Local Time Zone (TSLTZ)

Oracle9i JDBC drivers allow conversions among `DATE` and date/time datatypes. For example, you can access a `TIMESTAMPTZ` column as a `DATE` value.

Oracle9i 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 a time zone object because there are more Oracle time zone names than Sun has defined in its 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 classes are for the date/time data types:

- `oracle.sql.TIMESTAMP`

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

Use the following methods from the `oracle.jdbc.OraclePreparedStatement` interface to set a date/time item:

- `setTIMESTAMP(int paramIdx, TIMESTAMP x)`
- `setTIMESTAMPTZ(int paramIdx, TIMESTAMPTZ x)`
- `setTIMESTAMPLTZ(int paramIdx, TIMESTAMPLTZ x)`

Use the following methods from the `oracle.jdbc.OracleCallableStatement` interface to get a date/time item:

- `TIMESTAMP getTimeStamp (int paramIdx)`
- `TIMESTAMPTZ getTimeStampTZ(int paramIdx)`
- `TIMESTAMPLTZ getTimeStampLTZ(int paramIdx)`

Use the following methods from the `oracle.jdbc.OracleResultSet` interface to get a date/time item:

- `TIMESTAMP getTimeStamp(int paramIdx)`
- `TIMESTAMP getTimeStamp(java.lang.String colName)`
- `TIMESTAMPTZ getTimeStampTZ(int paramIdx)`
- `TIMESTAMPTZ getTimeStampTZ(java.lang.String colName)`
- `TIMESTAMPLTZ getTimeStampLTZ(int paramIdx)`
- `TIMESTAMPLTZ getTimeStampLTZ(java.lang.String colName)`
- `TIMESTAMPLTZ getTimeStampLTZ(int paramIdx)`

Before accessing `TIMESTAMPLTZ` data, call the `OracleConnection.setSessionTime()` 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 `TIMESTAMPLTZ` 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.

For information about accessing `ROWID` data, see "[Oracle ROWID Type](#)" on page 5-33.

Class `oracle.sql.OPAQUE`

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

Note: For Oracle9i 9.0.1, there is minimal support for `OPAQUE` types.

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.
- `public boolean 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` 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 (raw bytes).
- `toJdbc()`: Returns the JDBC representation of the `Datum` object.

Package oracle.jdbc

The interfaces of the `oracle.jdbc` package provide Oracle-specific extensions to allow access to raw SQL format data by using `oracle.sql.*` objects.

Note: The interfaces of the `oracle.jdbc` package replace the deprecated classes of the `oracle.jdbc.driver` package found in previous releases. (See "[Package oracle.jdbc](#)" on page 1-13 for more information.)

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
<code>OracleDriver</code>	Class	implements <code>java.sql.Driver</code>
<code>OracleConnection</code>	Interface	methods to return Oracle statement objects; methods to set Oracle performance extensions for any statement executed in the current connection (implements <code>java.sql.Connection</code>)
<code>OracleStatement</code>	Interface	methods to set Oracle performance extensions for individual statement; superclass of <code>OraclePreparedStatement</code> and <code>OracleCallableStatement</code> (implements <code>java.sql.Statement</code>)
<code>OraclePreparedStatement</code>	Interface	<code>setXXX()</code> methods to bind <code>oracle.sql.*</code> types into a prepared statement (implements <code>java.sql.PreparedStatement</code> ; extends <code>OracleStatement</code> ; superclass of <code>OracleCallableStatement</code>)

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

Name	Interface or Class	Key Functionality
OracleCallableStatement	Interface	getXXX() methods to retrieve data in oracle.sql format; setXXX() methods to bind oracle.sql.* types into a callable statement (implements java.sql.CallableStatement; extends OraclePreparedStatement)
OracleResultSet	Interface	getXXX() methods to retrieve data in oracle.sql format (implements java.sql.ResultSet)
OracleResultSetMetaData	Interface	methods to get meta information about Oracle result sets, such as column names and datatypes (implements java.sql.ResultSetMetaData)
OracleDatabaseMetaData	Class	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.

The remainder of this section describes the interfaces and classes of the oracle.jdbc package. For more information about using these interfaces and classes to access Oracle type extensions, see [Chapter 6, "Accessing and Manipulating Oracle Data"](#).

Class oracle.jdbc.OracleDriver

Use this class to register the Oracle JDBC drivers for use by your application. You can input a new instance of this class to the static registerDriver() method of the java.sql.DriverManager class so that your application can access and use the Oracle drivers. The registerDriver() method takes as input a "driver" class,

that is, a class that implements the `java.sql.Driver` interface, as is the case with `OracleDriver`.

Once you register the Oracle JDBC drivers, you can create your connection using the `DriverManager` class. For more information on registering drivers and writing a connection string, see ["First Steps in JDBC"](#) on page 3-2.

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, and support type maps for Oracle objects.

["Additional Oracle Performance Extensions"](#) on page 12-20 describes the performance extensions, including row prefetching, update batching, and metadata `TABLE_REMARKS` reporting.

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.

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

- `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.
- `getRemarksReporting()`: Returns true if `TABLE_REMARKS` reporting is enabled.
- `setRemarksReporting()`: Enables or disables `TABLE_REMARKS` reporting.

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.

"[Additional Oracle Performance Extensions](#)" on page 12-20 describes the performance extensions, including row prefetching and column type definitions.

Key methods include:

- `executeQuery()`: Executes a database query and returns an `OracleResultSet` object.
- `getResultSet()`: Retrieves an `OracleResultSet` object.
- `close()`: Closes the current statement.

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

- `defineColumnType()`: Defines the type you will use to retrieve data from a particular database table column.
- `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 `setXXX()` methods for binding `oracle.sql.*` types and objects into prepared statements, and methods to support Oracle performance extensions on a statement-by-statement basis.

"[Additional Oracle Performance Extensions](#)" on page 12-20 describes the performance extensions, including database update batching.

Key methods include:

- `getExecuteBatch()`: Retrieves the update-batching value for this statement.
- `setExecuteBatch()`: Sets the update-batching value for this statement.
- `setOracleObject()`: This is a generic `setXXX()` method for binding `oracle.sql.*` data into a prepared statement as an `oracle.sql.Datum` object.
- `setXXX()`: These methods, such as `setBLOB()`, are for binding specific `oracle.sql.*` types into prepared statements.
- `setORAData()`: Binds an `ORAData` object (for use in mapping Oracle object types to Java) into a prepared statement.
- `setNull()`: 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 (`schema.sql_type_name`) of the SQL type.
- `setFormOfUse()`: 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`, the JDBC driver will represent the provided data in the national character set of the server. The following code show how the `FORM_NCHAR` is used:

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

Key methods include:

- `getOracleObject()`: This is a generic `getXXX()` method for retrieving data into an `oracle.sql.Datum` object, which can be cast to the specific `oracle.sql.*` type as necessary.

- `getXXX()`: These methods, such as `getCLOB()`, are for retrieving data into specific `oracle.sql.*` objects.
- `setOracleObject()`: This is a generic `setXXX()` method for binding `oracle.sql.*` data into a callable statement as an `oracle.sql.Datum` object.
- `setXXX()`: These methods, such as `setBLOB()`, are inherited from `OraclePreparedStatement` for binding specific `oracle.sql.*` objects into callable statements.
- `setNull()`: 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 (`schema.type`) name of the SQL type.
- `setFormOfUse()`: 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`, the JDBC driver will represent the provided data in the national character set of the server. The following code show how the `FORM_NCHAR` is used:

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

- `registerOutParameter()`: Registers the SQL typecode of the statement's output parameter. 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 typecode 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()`: Closes the current result set, if any, and the current statement.

Interface `oracle.jdbc.OracleResultSet`

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

Key methods include:

- `getOracleObject()`: This is a generic `getXXX()` method for retrieving data into an `oracle.sql.Datum` object. It can be cast to the specific `oracle.sql.*` type as necessary.
- `getXXX()`: 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 ["Using Result Set Meta Data Extensions"](#) on page 6-19 for information on the functionality of the `OracleResultSetMetaData` interface.

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 typecode definitions of the standard Java `java.sql.Types` class and contains these additional typecodes for Oracle extensions:

- `OracleTypes.BFILE`
- `OracleTypes.ROWID`
- `OracleTypes.CURSOR` (for REF CURSOR types)

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

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 typecodes 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` interface and `oracle.jdbc.OracleCallableStatement` interface.

These are the forms that `registerOutputParameter()` can take for `CallableStatement` and `OracleCallableStatement` (assume a standard callable statement object `cs`):

```
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 typecode for the SQL datatype, *sql_name* is the name given to the datatype (for user-defined types, when *sqlType* is a STRUCT, REF, or ARRAY typecode), and *scale* represents the number of digits to the right of the decimal point (when *sqlType* is a NUMERIC or DECIMAL typecode).

Note: The second signature is standard under JDBC 2.0 in a JDK 1.2.x environment, but is an Oracle extension under JDK 1.1.x.

The following example uses a `CallableStatement` to call a procedure named `charout`, which returns a `CHAR` datatype. Note the use of the `OracleTypes.CHAR` typecode in the `registerOutParameter()` method (although `java.sql.Types.CHAR` could have been used as well).

```
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` datatype. The form of `registerOutParameter()` requires you to specify the typecode (`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` datatype. To retrieve the value of `EMPLOYEE` as an `oracle.sql.STRUCT` object, the statement object `cs` is cast to an `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 typecodes 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` interface and the `oracle.jdbc.OraclePreparedStatement` interface.

These are the forms that `setNull()` can take for `PreparedStatement` and `OraclePreparedStatement` objects (assume a standard prepared statement object `ps`):

```
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 typecode for the SQL datatype, and `sql_name` is the name given to the datatype (for user-defined types, when `sqlType` is a `STRUCT`, `REF`, or `ARRAY` typecode). If you enter an invalid `sqlType`, a `Parameter Type Conflict` exception is thrown.

Note: The second signature is standard under JDBC 2.0 in a JDK 1.2.x environment, but is an Oracle extension under JDK 1.1.x.

The following example uses a `PreparedStatement` to insert a `NULL` numeric value into the database. Note the use of `OracleTypes.NUMERIC` to identify the numeric object set to `NULL` (although `Types.NUMERIC` could have been used as well).

```
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 ();
```

Oracle Interfaces for Oracle-specific Features

The `oracle.jdbc` interfaces introduced in Oracle9i are recommended alternatives to the classes by the same name in the `oracle.jdbc.driver` package in older releases. These interfaces essentially duplicate the functionality in the `oracle.jdbc.driver` package.

The following example shows how the `oracle.jdbc` package is used to cast `pstmt` as an Oracle type:

```
java.sql.PreparedStatement pstmt
    = conn.prepareStatement(...);

((oracle.jdbc.OraclePreparedStatement) pstmt)
    .setExecuteBatch(10); // Oracle-specific method
```

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 datatype classes, the `getJavaSqlConnection()` method was added:

- `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. It throws a SQLException
//
oracle.jdbc.driver.OracleConnection
    getConnection() throws SQLException;

...
}
```

Package oracle.jdbc2 (for JDK 1.1.x only)

The `oracle.jdbc2` package is an Oracle implementation for use with JDK 1.1.x, containing classes and interfaces that mimic a subset of standard JDBC 2.0 classes and interfaces (which exist in the JDK 1.2 version of the standard `java.sql` package).

The following interfaces are implemented by `oracle.sql.*` type classes for JDBC 2.0-compliant Oracle type extensions under JDK 1.1.x.

- `oracle.jdbc2.Array` is implemented by `oracle.sql.ARRAY`
- `oracle.jdbc2.Struct` is implemented by `oracle.sql.STRUCT`
- `oracle.jdbc2.Ref` is implemented by `oracle.sql.REF`
- `oracle.jdbc2.Clob` is implemented by `oracle.sql.CLOB`
- `oracle.jdbc2.Blob` is implemented by `oracle.sql.BLOB`

In addition, the `oracle.jdbc2` package includes the following interfaces for users employing the JDBC-standard `SQLData` interface to create Java classes that map to Oracle objects. Again, these interfaces mimic `java.sql` interfaces available with JDK 1.2:

- `oracle.jdbc2.SQLData` is implemented by classes that map to Oracle objects; users must provide this implementation
- `oracle.jdbc2.SQLInput` is implemented by classes that read object data; Oracle provides a `SQLInput` class that the JDBC drivers use
- `oracle.jdbc2.SQLOutput` is implemented by classes that write object data; Oracle provides a `SQLOutput` class that the JDBC drivers use

The `SQLData` interface is one of the two facilities you can use to support Oracle objects in Java. (The other choice is the Oracle `ORADData` interface, included in the

oracle.sql package.) See "[Understanding the SQLData Interface](#)" on page 8-15 for more information about `SQLData`, `SQLInput`, and `SQLOutput`.

Oracle Character Datatypes Support

Oracle character datatypes include the SQL CHAR and SQL NCHAR datatypes. The following sections describe how these datatypes can be accessed using the Oracle JDBC drivers.

SQL CHAR Datatypes

The SQL CHAR datatypes include CHAR, VARCHAR2, and CLOB. These datatypes allow you to 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 Datatypes

SQL NCHAR datatypes were created for Globalization Support (formerly NLS). SQL NCHAR datatypes include NCHAR, NVARCHAR2, and NCLOB. These datatypes 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. See the *Oracle9i Globalization Support Guide* for information on SQL NCHAR datatypes.

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

The usage of SQL NCHAR datatypes is similar to that of the SQL CHAR (CHAR, VARCHAR2, and CLOB) datatypes. JDBC uses the same classes and methods to access SQL NCHAR datatypes that are used for the corresponding SQL CHAR datatypes. Therefore, there are no separate, corresponding classes defined in the `oracle.sql` package for SQL NCHAR datatypes. Likewise, there is no separate, corresponding constant defined in the `oracle.jdbc.OracleTypes` class for SQL NCHAR datatypes. The only difference in usage between the two datatypes 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 datatype.

Note: For Oracle9i 9.0.1, 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, FORM_NCHAR);
pstmt.setFormOfUse(3, FORM_NCHAR);

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

Class `oracle.sql.CHAR`

The `CHAR` class is used by Oracle JDBC in handling and converting character data. The JDBC driver constructs and populates `oracle.sql.CHAR` objects once character data has been read from the database.

Note: The `oracle.sql.CHAR` class is used for both SQL `CHAR` and SQL `NCHAR` datatypes.

The `CHAR` objects constructed and returned by the JDBC driver can be in the database character set, UTF-8, or ISO-Latin-1 (WE8ISO8859P1). 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, since the JDBC driver automatically creates `CHAR` objects as character data are obtained from the database. There may be circumstances, however, where constructing `CHAR` objects directly in application code is useful—for example, to repeatedly pass the same character data to one or more prepared statements without the overhead of converting from Java strings each time.

oracle.sql.CHAR Objects and Character Sets

The `CHAR` class provides Globalization Support functionality to convert character data. This class has two key attributes: (1) Globalization Support character set and (2) 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 being known, the bytes of data in the `CHAR` object are meaningless.

The `oracle.sql.CharacterSet` class is instantiated to represent character sets. 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. You can find a complete list of the character sets that Oracle supports in the *Oracle9i Globalization Support Guide*.

When constructing a `CHAR` object, there are limitations to which character sets you can use. If you are constructing a `CHAR` object for a unicode column (`NCHAR`, `NVARCHAR2`, and `NCLOB`), then use the database character set. If you are constructing a `CHAR` object for a `CHAR`, `VARCHAR`, or `VARCHAR2` column, use one of the character sets in [Table 5–3](#).

Table 5–3 Character Sets for SQL CHAR Datatypes

Character Set	Use if...
US7ASCII	the database character set is US7ASCII
WE8ISO8859P1	the database character set is WE8ISO8859P1
UTF8	the database character set is not US7ASCII or WE8ISO8859P1

If you are constructing a `CHAR` object for `NCHAR` datatypes, use the database `NCHAR` character set.

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.

Note: If you enter an invalid character set ID, an exception will *not* be thrown. Instead, you will receive unpredictable results.

For more information on character sets and character set IDs, see the *Oracle9i Globalization Support Guide*.

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—they 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.

See the `oracle.sql.CHAR` class Javadoc for more information.

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 server (a database) 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. For more information on how the JDBC drivers convert between character sets, see ["JDBC and Globalization Support"](#) on page 18-2.

Additional Oracle Type Extensions

See other chapters in this book for information about key Oracle type extensions:

- [Chapter 7, "Working with LOBs and BFILEs"](#)
- [Chapter 8, "Working with Oracle Object Types"](#)
- [Chapter 9, "Working with Oracle Object References"](#)
- [Chapter 10, "Working with Oracle Collections"](#)

This section covers additional Oracle type extensions and concludes with a discussion of differences between the current Oracle JDBC drivers and the Oracle 8.0.x and 7.3.x drivers regarding support of Oracle extensions.

Oracle JDBC drivers support the Oracle-specific `BFILE` and `ROWID` datatypes and `REF CURSOR` types, which were introduced in Oracle7 and are not part of the standard JDBC specification. This section describes the `ROWID` and `REF CURSOR` type extensions. See [Chapter 7](#) for information about `BFILEs`.

`ROWID` is supported as a Java string, and `REF CURSOR` types are supported as JDBC result sets.

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 type `ROWID` SQL data.

`ROWID`s 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 `ROWID`s with the result set `getString()` method (passing in either the column index or the column name). You can also bind a `ROWID` to a `PreparedStatement` parameter with the `setString()` method. This allows in-place updates, 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: ROWID 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 (address) 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 datatype `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 "datatype specifier" that identifies many different types of cursor variables.

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 datatypes, rather than a particular datatype.

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, `REF CURSOR`s are materialized as `ResultSet` objects 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: Accessing `REF CURSOR` Data 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.prepareStatement
    ("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 typecode `OracleTypes.CURSOR` for a REF CURSOR.
- The callable statement is executed, returning the REF CURSOR.
- The `CallableStatement` object is cast to an `OracleCallableStatement` object to use the `getCursor()` method, which is an Oracle extension to the standard JDBC API, and returns the REF CURSOR into a `ResultSet` object.

For a full sample application using a REF CURSOR, see ["REF CURSORS—RefCursorExample.java"](#) on page 20-38.

Support for Oracle Extensions in 8.0.x and 7.3.x JDBC Drivers

Some of the Oracle type extensions supported by the current Oracle JDBC drivers are either not supported or are supported differently by the Oracle 8.0.x and 7.3.x JDBC drivers. The following are the key points:

- The 8.0.x and 7.3.x drivers have no `oracle.sql` package, meaning there are no wrapper types such as `oracle.sql.NUMBER` and `oracle.sql.CHAR` that you can use to wrap raw SQL data.
- The 8.0.x and 7.3.x drivers do not support Oracle object and collection types.
- The 8.0.x and 7.3.x drivers support the Oracle ROWID datatype with the `OracleRowid` class in the `oracle.jdbc` package.

- The 8.0.x drivers support the Oracle BLOB, CLOB, and BFILE datatypes with the `OracleBlob`, `OracleClob`, and `OracleBfile` classes in the `oracle.jdbc` package. These classes do not include LOB and BFILE manipulation methods—you must instead use the PL/SQL `DBMS_LOB` package.
- The 7.3.x drivers do not support BLOB, CLOB, and BFILE.

[Table 5–4](#) summarizes these differences. "OracleTypes Definition" refers to static typecode constants defined in the `oracle.jdbc.OracleTypes` class.

Table 5–4 Support for Oracle Type Extensions, 8.0.x and 7.3.x JDBC Drivers

Oracle Datatype	OracleTypes Definition	Type Extension, Current Drivers	Type Extension, 8.0.x/7.3.x drivers
NUMBER	OracleTypes.NUMBER	oracle.sql.NUMBER	no type extension for wrapper class
CHAR	OracleTypes.CHAR	oracle.sql.CHAR	no type extension for wrapper class
RAW	OracleTypes.RAW	oracle.sql.RAW	no type extension for wrapper class
DATE	OracleTypes.DATE	oracle.sql.DATE	no type extension for wrapper class
ROWID	OracleTypes.ROWID	oracle.sql.ROWID	oracle.jdbc.driver.OracleRowid
BLOB	OracleTypes.BLOB	oracle.sql.BLOB	oracle.jdbc.driver.OracleBlob in 8.0.x; not supported in 7.3.x
CLOB	OracleTypes.CLOB	oracle.sql.CLOB	oracle.jdbc.driver.OracleClob in 8.0.x; not supported in 7.3.x
BFILE	n/a	oracle.sql.BFILE	oracle.jdbc.driver.OracleBfile in 8.0.x; not supported in 7.3.x
structured object	OracleTypes.STRUCT	oracle.sql.STRUCT or custom class	not supported
object reference	OracleTypes.REF	oracle.sql.REF or custom class	not supported
collection (array)	OracleTypes.ARRAY	oracle.sql.ARRAY or custom class	not supported
OPAQUE	OracleTypes.OPAQUE	oracle.sql.OPAQUE	not supported

Accessing and Manipulating Oracle Data

This chapter describes data access in `oracle.sql.*` formats, as opposed to standard Java formats. As described in the previous chapter, the `oracle.sql.*` formats are a key factor of the Oracle 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` objects, as appropriate, and using the `getOracleObject()`, `setOracleObject()`, `getXXX()`, and `setXXX()` methods of these classes (where `XXX` corresponds to the types in the `oracle.sql` package).

This chapter covers the following topics:

- [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 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. The classes in this package simply wrap the raw SQL data.

Standard Types versus Oracle Types

In processing speed and effort, the `oracle.sql.*` classes provide the most efficient way of representing SQL data. These classes store the usual representations of SQL data as byte arrays. They do not reformat the data or perform any character-set conversions (aside from the usual network conversions) on it. The data remains in SQL format, and therefore no information is lost. For SQL primitive types (such as `NUMBER`, and `CHAR`), the `oracle.sql.*` classes simply wrap the SQL data. For SQL structured types (such as objects and arrays), the classes provide additional information such as conversion methods and structure details.

If you are moving data within the database, then you will probably want to keep your data in `oracle.sql.*` format. If you are displaying the data or performing calculations on it in a Java application running outside the database, then you will probably want to materialize the data as instances of standard types such as `java.sql.*` or `java.lang.*` types. Similarly, if you are using a parser that expects the data to be in a standard Java format, then you must use one of the standard formats instead of `oracle.sql.*` format.

Converting SQL NULL Data

Java represents a SQL `NULL` datum by the Java value `null`. Java datatypes fall into two categories: primitive types (such as `byte`, `int`, `float`) and object types (class instances). The primitive types cannot represent `null`. Instead, they store the 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 (for example, `Integer` for `int`, `Float` for `float`) that can represent `null`. The object wrapper types must be used as the targets for SQL data to detect SQL `NULL` without ambiguity.

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, keep it as a `ResultSet` type. However, if you want to use the Oracle extensions on the object, you must cast it to an `OracleResultSet` type. Although the type by which the Java compiler will identify the object is changed, the object itself is unchanged.

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` object, as above, and then cast that object into an `OracleResultSet` object later.

Similarly, when you want to execute a stored procedure using a callable statement, the JDBC drivers will return an `OracleCallableStatement` object typed as a `java.sql.CallableStatement`. If you want to apply only standard JDBC methods to the object, then keep it as a `CallableStatement` type. However, if you want to use the Oracle extensions on the object, you must cast it to an `OracleCallableStatement` type. Although the type by which the Java compiler will identify the object is changed, the object itself is unchanged.

You use the standard JDBC `java.sql.Connection.prepareStatement()` method to create a `PreparedStatement` object. If you want to apply only standard JDBC methods to the object, keep it as a `PreparedStatement` type. However, if you want to use the Oracle extensions on the object, you must cast it to an `OraclePreparedStatement` type. While the type by which the Java compiler will identify the object is changed, the object itself is unchanged.

Key extensions to the result set and statement classes include `getOracleObject()` and `setOracleObject()` methods that you can use to access and manipulate data in `oracle.sql.*` formats, instead of standard Java formats. For more information, see the next section: "[Comparison of Oracle get and set Methods to Standard JDBC](#)".

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 `getXXX()` methods for all the Oracle SQL types (as described in "[Other getXXX\(\) Methods](#)" on page 6-7), 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.

Standard `getObject()` Method

The standard JDBC `getObject()` method of a result set or callable statement returns data into a `java.lang.Object` object. The format of the data returned is based on its original type, as follows:

- For SQL datatypes that are not Oracle-specific, `getObject()` returns the default Java type corresponding to the column's SQL type, following the mapping specified in the JDBC specification.
- For Oracle-specific datatypes (such as ROWID, discussed in "[Oracle ROWID Type](#)" on page 5-33), `getObject()` returns an object of the appropriate `oracle.sql.*` class (such as `oracle.sql.ROWID`).
- For Oracle objects, `getObject()` returns an object of the Java class specified in your type map. (Type maps specify the correlation between Java classes and database SQL types and are discussed in "[Understanding Type Maps for SQLData Implementations](#)" on page 8-11.) The `getObject(parameter_index)` method uses the connection's default type map. 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.

For more information on `getObject()` return types, see [Table 6-1, "Summary of getObject\(\) and getOracleObject\(\) Return Types"](#) on page 6-6.

Oracle `getOracleObject()` Method

If you want to retrieve data from a result set or callable statement into an `oracle.sql.*` object, then cast your result set to an `OracleResultSet` type or your callable statement to an `OracleCallableStatement` type, and use the `getOracleObject()` method.

When you use `getOracleObject()`, the data will be of the appropriate `oracle.sql.*` type and is returned into an `oracle.sql.Datum` object (the `oracle.sql` type classes extend `Datum`). The signature for the method is:

```
public oracle.sql.Datum getOracleObject(int parameter_index)
```

When you have retrieved data into a `Datum` object, you can use the standard Java `instanceof` operator to determine which `oracle.sql.*` type it really is.

For more information on `getOracleObject()` return types, see [Table 6-1, "Summary of getObject\(\) and getOracleObject\(\) Return Types"](#) on page 6-6.

Example: Using getOracleObject() with a ResultSet The following example creates a table that contains a column of character data (in this case, a row number) and a column containing a `BFILE` locator. A `SELECT` statement retrieves the contents of the table into 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 results 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 (in this case a name) with a date. The program passes the string `SCOTT` to the prepared call and registers the `DATE` type as an output parameter. After the call is executed, `getOracleObject()` retrieves the date associated with the name `SCOTT`. Note that because `getOracleObject()` returns a `Datum` object, the results are cast to a `DATE` object.

```
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 6-1](#) summarizes the information in the preceding sections, "[Standard getObject\(\) Method](#)" and "[Oracle getOracleObject\(\) Method](#)" on page 6-4.

This table lists the underlying return types for each method for each Oracle SQL type, but keep in mind the signatures of the methods when you write your code:

- getObject(): Always returns data into a java.lang.Object instance.
- getOracleObject(): Always returns data into an oracle.sql.Datum instance.

You must cast the returned object to use any special functionality (see "[Casting Your get Method Return Values](#)" on page 6-10).

Table 6-1 Summary of 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
NUMBER	java.math.BigDecimal	oracle.sql.NUMBER
RAW	byte[]	oracle.sql.RAW
LONGRAW	byte[]	oracle.sql.RAW
DATE	java.sql.Timestamp	oracle.sql.DATE
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

Table 6–1 Summary of getObject() and getOracleObject() Return Types (Cont.)

Oracle SQL Type	getObject() Underlying Return Type	getOracleObject() Underlying Return Type
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

For information on type compatibility between all SQL and Java types, see [Table 21–1, "Valid SQL Datatype-Java Class Mappings"](#) on page 21-2.

Other getXXX() Methods

Standard JDBC provides a `getXXX()` for each standard Java type, such as `getBytes()`, `getInt()`, `getFloat()`, and so on. Each of these returns exactly what the method name implies (a byte, an int, a float, and so on).

In addition, the `OracleResultSet` and `OracleCallableStatement` classes provide a full complement of `getXXX()` methods corresponding to all the `oracle.sql.*` types. Each `getXXX()` method returns an `oracle.sql.XXX` object. For example, `getROWID()` returns an `oracle.sql.ROWID` object.

Some of these extensions are taken from the JDBC 2.0 specification. They return objects of type `java.sql.*` (or `oracle.jdbc2.*` under JDK 1.1.x), instead of `oracle.sql.*`. For example, compare the following method names and return types:

```
java.sql.Blob getBlob(int parameter_index)
```

```
oracle.sql.BLOB getBLOB(int parameter_index)
```

Although there is no particular performance advantage in using the specific `getXXX()` methods, they can save you the trouble of casting, because they return specific object types.

Return Types and Input Parameter Types of getXXX() Methods

[Table 6–2](#) summarizes the underlying return types and the input parameter types for each `getXXX()` method, and notes which are Oracle extensions under JDK 1.2.x

and JDK 1.1.x. You must cast to an `OracleResultSet` or `OracleCallableStatement` to use methods that are Oracle extensions.

Table 6–2 Summary of `getXXX()` Return Types

Method	Underlying Return Type	Signature Type	Oracle Ext for JDK 1.2.x?	Oracle Ext for JDK 1.1.x?
<code>getArray()</code>	<code>oracle.sql.ARRAY</code>	<code>java.sql.Array</code> (<code>oracle.jdbc2.Array</code> under JDK 1.1.x)	No	Yes
<code>getARRAY()</code>	<code>oracle.sql.ARRAY</code>	<code>oracle.sql.ARRAY</code>	Yes	Yes
<code>getAsciiStream()</code>	<code>java.io.InputStream</code>	<code>java.io.InputStream</code>	No	No
<code>getBfile()</code>	<code>oracle.sql.BFILE</code>	<code>oracle.sql.BFILE</code>	Yes	Yes
<code>getBFILE()</code>	<code>oracle.sql.BFILE</code>	<code>oracle.sql.BFILE</code>	Yes	Yes
<code>getBigDecimal()</code> (see Notes section below)	<code>java.math.BigDecimal</code>	<code>java.math.BigDecimal</code>	No	No
<code>getBinaryStream()</code>	<code>java.io.InputStream</code>	<code>java.io.InputStream</code>	No	No
<code>getBlob()</code>	<code>oracle.sql.BLOB</code>	<code>java.sql.Blob</code> (<code>oracle.jdbc2.Blob</code> under JDK 1.1.x)	No	Yes
<code>getBLOB</code>	<code>oracle.sql.BLOB</code>	<code>oracle.sql.BLOB</code>	Yes	Yes
<code>getBoolean()</code>	<code>boolean</code>	<code>boolean</code>	No	No
<code>getByte()</code>	<code>byte</code>	<code>byte</code>	No	No
<code>getBytes()</code>	<code>byte[]</code>	<code>byte[]</code>	No	No
<code>getCHAR()</code>	<code>oracle.sql.CHAR</code>	<code>oracle.sql.CHAR</code>	Yes	Yes
<code>getCharacterStream()</code>	<code>java.io.Reader</code>	<code>java.io.Reader</code>	No	Yes
<code>getClob()</code>	<code>oracle.sql.CLOB</code>	<code>java.sql.Clob</code> (<code>oracle.jdbc2.Clob</code> under JDK 1.1.x)	No	Yes
<code>getCLOB()</code>	<code>oracle.sql.CLOB</code>	<code>oracle.sql.CLOB</code>	Yes	Yes
<code>getDate()</code> (see Notes section below)	<code>java.sql.Date</code>	<code>java.sql.Date</code>	No	No
<code>getDATE()</code>	<code>oracle.sql.DATE</code>	<code>oracle.sql.DATE</code>	Yes	Yes

Table 6–2 Summary of getXXX() Return Types (Cont.)

Method	Underlying Return Type	Signature Type	Oracle Ext for JDK 1.2.x?	Oracle Ext for JDK 1.1.x?
getDouble()	double	double	No	No
getFloat()	float	float	No	No
getInt()	int	int	No	No
getLong()	long	long	No	No
getNUMBER()	oracle.sql.NUMBER	oracle.sql.NUMBER	Yes	Yes
getOracleObject()	subclasses of oracle.sql.Datum	oracle.sql.Datum	Yes	Yes
getRAW()	oracle.sql.RAW	oracle.sql.RAW	Yes	Yes
getRef()	oracle.sql.REF	java.sql.Ref (oracle.jdbc2.Ref under JDK 1.1.x)	No	Yes
getREF()	oracle.sql.REF	oracle.sql.REF	Yes	Yes
getROWID()	oracle.sql.ROWID	oracle.sql.ROWID	Yes	Yes
getShort()	short	short	No	No
getString()	String	String	No	No
getSTRUCT()	oracle.sql.STRUCT.	oracle.sql.STRUCT	Yes	Yes
getTime() (see Notes section below)	java.sql.Time	java.sql.Time	No	No
getTimestamp() (see Notes section below)	java.sql.Timestamp	java.sql.Timestamp	No	No
getUnicodeStream()	java.io.InputStream	java.io.InputStream	No	No

Special Notes about getXXX() Methods

This section provides additional details about some of the `getXXX()` methods.

getBigDecimal() Note

JDBC 2.0 supports a simplified method signature for the `getBigDecimal()` method. The previous input signature was:

`(int columnIndex, int scale)` or `(String columnName, int scale)`

The new input signature is simply:

`(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.

getDate(), getTime(), and getTimestamp() Note

In JDBC 2.0, the `getDate()`, `getTime()`, and `getTimestamp()` methods have the following input signatures:

`(int columnIndex, Calendar cal)`

or:

`(String columnName, Calendar cal)`

The Oracle JDBC drivers ignore the `Calendar` object input, because it is not currently feasible to support `java.sql.Date` timezone information together with the data. You should continue to use previous input signatures that take only the column index or column name. `Calendar` input will be supported in a future Oracle JDBC release.

Casting Your get Method Return Values

As described in "[Standard getObject\(\) Method](#)" on page 6-4, Oracle's implementation of `getObject()` always returns a `java.lang.Object` instance, and `getOracleObject()` always returns an `oracle.sql.Datum` instance. Usually, you would cast the returned object to the appropriate class so that you could use particular methods and functionality of that class.

In addition, you have the option of using a specific `getXXX()` method instead of the generic `getObject()` or `getOracleObject()` methods. The `getXXX()` methods enable you to avoid casting, because the return type of `getXXX()` corresponds to the type of object returned. For example, `getCLOB()` returns an `oracle.sql.CLOB` instance, as opposed to a `java.lang.Object` instance.

Example: Casting Return Values This example assumes that you have fetched data of type `CHAR` into a result set (where it is in column 1). Because you want to manipulate the `CHAR` data without losing precision, cast your result set to an `OracleResultSet`, and use `getOracleObject()` to return the `CHAR` data in `oracle.sql.*` format. If you do not cast your result set, you have to use

`getObject()`, which returns your character data into a Java `String` and loses some of the precision of your SQL data.

The `getOracleObject()` method returns an `oracle.sql.CHAR` object into an `oracle.sql.Datum` return variable unless you cast the output. Cast the `getOracleObject()` output to `oracle.sql.CHAR` if you want to use a `CHAR` return variable and any of the special functionality of that class (such as the `getCharacterSet()` method that returns the character set used to represent the characters).

```
CHAR char = (CHAR)ors.getOracleObject(1);
CharacterSet cs = char.getCharacterSet();
```

Alternatively, you can return the object into a generic `oracle.sql.Datum` return variable and cast it later when you must use the `CHAR` `getCharacterSet()` method.

```
Datum rawdatum = ors.getOracleObject(1);
...
CharacterSet cs = ((CHAR)rawdatum).getCharacterSet();
```

This uses the `getCharacterSet()` method of `oracle.sql.CHAR`. The `getCharacterSet()` method is not defined on `oracle.sql.Datum` and would not be reachable without the cast.

Standard `setObject()` and Oracle `setOracleObject()` Methods

Just as there is a standard `getObject()` and Oracle-specific `getOracleObject()` in result sets and callable statements for retrieving data, there is also a standard `setObject()` and an Oracle-specific `setOracleObject()` in Oracle prepared statements and callable statements for updating data. 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—it has been implemented so that you can also input instances of the `oracle.sql.*` classes that correspond to JDBC 2.0-compliant Oracle extensions: `BLOB`, `CLOB`, `BFILE`, `STRUCT`, `REF`, and `ARRAY`.

To bind `oracle.sql.*` types to a prepared statement or callable statement, use the `setOracleObject()` method, which takes an `oracle.sql.Datum` (or any subclass) as input. To use `setOracleObject()`, you must cast your prepared

statement or callable statement to an `OraclePreparedStatement` or `OracleCallableStatement` object.

Example: Using `setObject()` and `setOracleObject()` in a Prepared Statement This example assumes that you have fetched character data into a standard result set (where it is in column 1), and you want to cast the results to an `OracleResultSet` so that you can use Oracle-specific formats and methods. Because you want to use the data as `oracle.sql.CHAR` format, cast the results of the `getOracleObject()` (which returns type `oracle.sql.Datum`) to `CHAR`. Similarly, because you want to manipulate the data in column 2 as strings, cast the data to a Java `String` type (because `getObject()` returns data of type `Object`). In this example, `rs` represents the result set, `charVal` represents the data from column 1 in `oracle.sql.CHAR` format, and `strVal` represents the data from column 2 in Java `String` format.

```
CHAR charVal=(CHAR)((OracleResultSet)rs).getOracleObject(1);
String strVal=(String)rs.getObject(2);
...
```

For a prepared statement object `ps`, 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 an `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 `getXXX()` methods, there are several specific `setXXX()` methods. Standard `setXXX()` methods are provided for binding standard Java types, and Oracle-specific `setXXX()` methods are provided for binding Oracle-specific types.

Note: Under JDK 1.1.x, for compatibility with the JDBC 2.0 standard, `OraclePreparedStatement` and `OracleCallableStatement` classes provide `setXXX()` methods that take `oracle.jdbc2` input parameters for BLOBs, CLOBs, object references, and arrays. For example, a `setBlob()` method takes an `oracle.jdbc2.Blob` input parameter, where it would take a `java.sql.Blob` input parameter under JDK 1.2.x.

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 typecode 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. Under JDK 1.1.x, it is available as an Oracle extension. It takes a SQL type name in addition to a parameter index and a SQL type code. Use this method when the SQL typecode is `java.sql.Types.REF`, `ARRAY`, or `STRUCT`. (If the typecode 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)
```

For binding Oracle-specific types, using the appropriate specific `setXXX()` methods instead of methods for binding standard Java types may offer some performance advantage.

Input Parameter Types of `setXXX()` Methods

Table 6–3 summarizes the input types for all the `setXXX()` methods and notes which are Oracle extensions under JDK 1.2.x and JDK 1.1.x. To use methods that are Oracle extensions, you must cast your statement to an `OraclePreparedStatement` or `OracleCallableStatement`.

Table 6–3 Summary of `setXXX()` Input Parameter Types

Method	Input Parameter Type	Oracle Ext for JDK 1.2.x?	Oracle Ext for JDK 1.1.x?
<code>setArray()</code>	<code>java.sql.Array</code> (<code>oracle.jdbc2.Array</code> under JDK 1.1.x)	No	Yes
<code>setARRAY()</code>	<code>oracle.sql.ARRAY</code>	Yes	Yes

Table 6–3 Summary of setXXX() Input Parameter Types (Cont.)

Method	Input Parameter Type	Oracle Ext for JDK 1.2.x?	Oracle Ext for JDK 1.1.x?
setAsciiStream() (see Notes section below)	java.io.InputStream	No	No
setBfile()	oracle.sql.BFILE	Yes	Yes
setBFILE()	oracle.sql.BFILE	Yes	Yes
setBigDecimal()	BigDecimal	No	No
setBinaryStream() (see Notes section below)	java.io.InputStream	No	No
setBlob()	java.sql.Blob (oracle.jdbc2.Blob under JDK 1.1.x)	No	Yes
setBLOB()	oracle.sql.BLOB	Yes	Yes
setBoolean()	boolean	No	No
setByte()	byte	No	No
setBytes()	byte[]	No	No
setCHAR() (also see setFixedCHAR() method)	oracle.sql.CHAR	Yes	Yes
setCharacterStream() (see Notes section below)	java.io.Reader	No	Yes
setClob()	java.sql.Clob (oracle.jdbc2.Clob under JDK 1.1.x)	No	Yes
setCLOB()	oracle.sql.CLOB	Yes	Yes
setDate() (see Notes section below)	java.sql.Date	No	No
setDATE()	oracle.sql.DATE	Yes	Yes
setDouble()	double	No	No
setFixedCHAR() (see setFixedCHAR() section below)	java.lang.String	Yes	Yes
setFloat()	float	No	No

Table 6–3 Summary of setXXX() Input Parameter Types (Cont.)

Method	Input Parameter Type	Oracle Ext for JDK 1.2.x?	Oracle Ext for JDK 1.1.x?
setInt()	int	No	No
setLong()	long	No	No
setNUMBER()	oracle.sql.NUMBER	Yes	Yes
setRAW()	oracle.sql.RAW	Yes	Yes
setRef()	java.sql.Ref (oracle.jdbc2.Ref under JDK 1.1.x)	No	Yes
setREF()	oracle.sql.REF	Yes	Yes
setROWID()	oracle.sql.ROWID	Yes	Yes
setShort()	short	No	No
setString()	String	No	No
setSTRUCT()	oracle.sql.STRUCT	Yes	Yes
setTime() (see note below)	java.sql.Time	No	No
setTimestamp() (see note below)	java.sql.Timestamp	No	No
setUnicodeStream() (see note below)	java.io.InputStream	No	No

For information on all supported type mappings between SQL and Java, see [Table 21–1, "Valid SQL Datatype-Java Class Mappings"](#) on page 21-2.

Setter Method Size Limitations on Oracle8 and Oracle7

[Table 6–4](#) lists size limitations for the `setBytes()` and `setString()` methods for SQL binds to Oracle8 and Oracle7 databases. (These limitations do not apply to PL/SQL binds.) For information about how to work around these limits using the stream API, see ["Using Streams to Avoid Limits on setBytes\(\) and setString\(\)"](#) on page 3-30.

Table 6–4 Size Limitations for *setBytes()* and *setString()* Methods

	Oracle8	Oracle7
setBytes() size limitation	2000 bytes	255 bytes
setString() size limitation	4000 bytes	2000 bytes

Setter Methods That Take Additional Input

The following `setXXX()` 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.

Important: The preceding stream methods can also be used for LOBs. See "[Reading and Writing BLOB and CLOB Data](#)" on page 7-6 for more information.

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

- `setTimestamp(int paramIndex, Timestamp x, Calendar cal)`

The JDBC 2.0 signatures for `setDate()`, `setTime()`, and `setTimestamp()` include a `Calendar` object, but the Oracle JDBC drivers ignore this input because it is not yet feasible to support `java.sql.Date` timezone information together with the data. You should continue to use the previous signatures that take only the parameter index and data item. `Calendar` input will be supported in a future release.

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 executes a non-padded comparison.

Notes:

- 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 (coll char(10));
insert into my_table values ('JDBC');
*/
PreparedStatement pstmt = conn.prepareStatement
    ("select count(*) from my_table where coll = ?");

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;
}
```

Limitations of the Oracle 8.0.x and 7.3.x JDBC Drivers

The Oracle 8.0.x JDBC drivers use the same protocol as the Oracle 7.3.x JDBC drivers. In both cases, Oracle datatypes are as defined for an Oracle 7.3.x database, and data items longer than 2K bytes must be LONG.

As with any LONG data, use the stream APIs to read and write data between your application and the database. Essentially, this means that you cannot use the normal `getString()` and `setString()` methods to read or write data longer than 2K bytes when using the 8.0.x and 7.3.x drivers.

The stream APIs include methods such as `getBinaryStream()`, `setBinaryStream()`, `getAsciiStream()`, and `setAsciiStream()`. These methods are discussed under "[Java Streams in JDBC](#)" on page 3-19.

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 underlying protocol 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 getColumnName(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 each column's numerical type and SQL type name.

```
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.getColumnName (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
```

Working with LOBs and BFILEs

This chapter describes how you use JDBC and the `oracle.sql.*` classes to access and manipulate LOB and BFILE locators and data, covering the following topics:

- [Oracle Extensions for LOBs and BFILEs](#)
- [Working with BLOBs and CLOBs](#)
- [Working with BFILEs](#)

Oracle Extensions for LOBs and BFILES

LOBs ("large objects") are stored in a way that optimizes space and provides efficient access. The JDBC drivers provide support for two types of LOBs: BLOBs (unstructured binary data) and CLOBs (character data). BLOB and CLOB data is accessed and referenced by using a locator, which 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 following sections also describe how to create and populate a LOB column in a table.

The JDBC drivers support these `oracle.sql.*` classes for BLOBs, CLOBs, and BFILES:

- `oracle.sql.BLOB`
- `oracle.sql.CLOB`
- `oracle.sql.BFILE`

The `oracle.sql.BLOB` and `CLOB` classes implement the `java.sql.Blob` and `Clob` interfaces, respectively (`oracle.jdbc2.Blob` and `Clob` interfaces under JDK 1.1.x). By contrast, `BFILE` is an Oracle extension, without a corresponding `java.sql` (or `oracle.jdbc2`) interface.

Instances of these classes contain only the locators for these datatypes, not the data. After accessing the locators, you must perform some additional steps to access the data. These steps are described in "[Reading and Writing BLOB and CLOB Data](#)" on page 7-6 and "[Reading BFILE Data](#)" on page 7-22.

Note: You cannot construct `BLOB`, `CLOB`, or `BFILE` objects in your JDBC application—you can only retrieve existing `BLOBs`, `CLOBs`, or `BFILES` from the database or create them using the `createTemporary()` and `empty_lob()` methods.

Working with BLOBs and CLOBs

This section describes how to read and write data to and from binary large objects (BLOBs) and character large objects (CLOBs) in an Oracle database, using LOB locators.

For general information about Oracle9i LOBs and how to use them, see the *Oracle9i Application Developer's Guide - Large Objects (LOBs)*.

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.

Retrieving BLOB and CLOB Locators

Given a standard JDBC result set (`java.sql.ResultSet`) or callable statement (`java.sql.CallableStatement`) that includes BLOB or CLOB locators, you can access the locators by using standard getter methods, as follows. All the standard and Oracle-specific getter methods discussed here take either an `int` column index or a `String` column name as input.

- Under JDK 1.2.x, you can use the standard `getBlob()` and `getClob()` methods, which return `java.sql.Blob` and `Clob` objects, respectively.
- Under JDK 1.1.x, there is no standard BLOB or CLOB functionality, but you can use the generic `getObject()` method, which returns `java.lang.Object`, and cast the output as desired.

If you retrieve or cast the result set or callable statement to an `OracleResultSet` or `OracleCallableStatement` object, then you can use Oracle extensions as follows:

- Under either JDK 1.2.x or JDK 1.1.x, you can use `getBLOB()` and `getCLOB()`, which return `oracle.sql.BLOB` and `CLOB` objects, respectively.
- Under either JDK 1.2.x or JDK 1.1.x, you can also use the `getOracleObject()` method, which returns an `oracle.sql.Datum` object, and cast the output appropriately.
- Under JDK 1.1.x, you also have the option of using the Oracle extensions `getBlob()` and `getClob()`, which return `oracle.jdbc2.Blob` and `Clob` objects, respectively. (These `Blob` and `Clob` interfaces mimic the standard interfaces available in JDK 1.2.x.)

Note: If using `getObject()` or `getOracleObject()`, then remember to cast the output, as necessary. For more information, see ["Casting Your get Method Return Values"](#) on page 6-10.

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 `Clob`. As an alternative, you can cast the output to `oracle.sql.BLOB` and `CLOB` to take advantage of extended functionality offered by the `oracle.sql.*` classes. For example, you can rewrite the above 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` `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 typecode 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 (`java.sql.PreparedStatement`) or callable statement (`java.sql.CallableStatement`), you can use standard setter methods to pass LOB locators, as follows. All the standard and Oracle-specific setter methods discussed here take an `int` parameter index and the LOB locator as input.

- Under JDK 1.2.x, you can use the standard `setBlob()` and `setClob()` methods, which take `java.sql.Blob` and `Clob` locators as input.
- Under JDK 1.1.x, there is no standard BLOB or CLOB functionality, but you can use the generic `setObject()` method, which simply specifies a `java.lang.Object` input.

Given an Oracle-specific `OraclePreparedStatement` or `OracleCallableStatement`, then you can use Oracle extensions as follows:

- Under either JDK 1.2.x or JDK 1.1.x, you can use `setBLOB()` and `setCLOB()`, which take `oracle.sql.BLOB` and `CLOB` locators as input, respectively.
- Under either JDK 1.2.x or JDK 1.1.x, you can also use the `setOracleObject()` method, which simply specifies an `oracle.sql.Datum` input.
- Under JDK 1.1.x, you also have the option of using the Oracle extensions `setBlob()` and `setClob()`, which take `oracle.jdbc2.Blob` and `Clob` locators as input, respectively. (These `Blob` and `Clob` interfaces mimic the standard interfaces available in JDK 1.2.x.)

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, disable auto-commit mode.
 - The implementation of the data access API uses direct native calls in the JDBC OCI and server-side internal drivers, thereby providing better performance. You can use the same API on the LOB classes in all Oracle JDBC drivers.
 - In the case of the JDBC Thin driver only, the implementation of the data access API uses the PL/SQL `DBMS_LOB` package internally. You never have to use `DBMS_LOB` directly. This is in contrast to the 8.0.x drivers. For more information on the `DBMS_LOB` package, see the *Oracle9i Supplied PL/SQL Packages Reference*.
-
-

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 `getBinaryOutputStream()` 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 `getAsciiOutputStream()` or `getCharacterOutputStream()` method of an `oracle.sql.CLOB` object to retrieve the CLOB as an output stream to be written back to the CLOB. The `getAsciiOutputStream()` method returns an ASCII output stream in a `java.io.OutputStream` object. The `getCharacterOutputStream()` method returns a Unicode output stream in a `java.io.Writer` object.

As with any `OutputStream` 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 execute an `UPDATE` to write the data.
 - 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.
-
-

Important: The JDBC 2.0 specification states that `PreparedStatement` methods `setBinaryStream()` and `setObject()` can be used to input a stream value as a BLOB, and that the `PreparedStatement` methods `setAsciiStream()`, `setUnicodeStream()`, `setCharacterStream()`, and `setObject()` can be used to input a stream value as a CLOB. This bypasses the LOB locator, going directly to the LOB data itself.

In the implementation of the Oracle JDBC drivers, this functionality is supported *only* for a configuration using an 8.1.6 and higher database and 8.1.6 and higher JDBC OCI driver. **Do not use this functionality for any other configuration, as data corruption may result.**

Example: Reading BLOB Data Use the `getBinaryStream()` method of the `oracle.sql.BLOB` class to read BLOB data. The `getBinaryStream()` method reads the BLOB data into a binary stream.

The following example uses the `getBinaryStream()` method to read BLOB data into 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();  
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();
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 into 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();
byte[] asciiChar_array = new byte[10];
int asciiChar_read = asciiChar_stream.read(asciiChar_array,0,10);
```

Example: Writing BLOB Data Use the `getBinaryOutputStream()` 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 `getBinaryOutputStream()` 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).getBinaryOutputStream();
outstream.write(data);
...
```

Example: Writing CLOB Data Use the `getCharacterOutputStream()` method or the `getAsciiOutputStream()` method to write data to a CLOB. The `getCharacterOutputStream()` method returns a Unicode output stream; the `getAsciiOutputStream()` method returns an ASCII output stream.

The following example reads a vector of data into a character array, then uses the `getCharacterOutputStream()` method to write the array of character data to a CLOB. The `getCharacterOutputStream()` method returns a `java.io.Writer` instance in an `oracle.sql.CLOB` object, not a `java.sql.Clob` object.

```
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).getCharacterOutputStream();
writer.write(data);
writer.flush();
writer.close();

...
```

The next example reads a vector of data into a byte array, then uses the `getAsciiOutputStream()` method to write the array of ASCII data to a CLOB. Because `getAsciiOutputStream()` returns an ASCII output stream, you must cast the output to a `oracle.sql.CLOB` datatype.

```
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)clob).getAsciiOutputStream();
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, creating a file handler for the data (if you are reading the data from a file), 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, execute 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 GIF file `john.gif` to a BLOB.

1. Begin by using SQL statements to create the BLOB entry in the table. Use the `empty_blob` syntax 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'";
ResultSet rest = stmt.executeQuery(cmd);
BLOB blob = ((OracleResultSet)rset).getBLOB(2);
```

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 GIF 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.getBinaryOutputStream();
```

4. Call `getBufferSize()` to retrieve the ideal buffer size (according to calculations by the JDBC driver) 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 GIF 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.

```
while ((length = instream.read(buffer)) != -1)
    outstream.write(buffer, 0, length);
instream.close();
outstream.close();
```

Once your data is in the BLOB or CLOB, you can manipulate the data. This is described in the next section, "[Accessing and Manipulating BLOB and CLOB 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. "[Getting and Passing BLOB and CLOB Locators](#)" on page 7-3 describes these techniques in detail.

After you select the locators, you can retrieve the BLOB or CLOB data. You will usually want to cast the result set to the `OracleResultSet` datatype 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);
printBytes(bytes, length);
```

Additional BLOB and CLOB Features

In addition to what has already been discussed in this chapter, the `oracle.sql.BLOB` and `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.

Additional BLOB Methods

The `oracle.sql.BLOB` class includes the following methods:

- `close()`: Closes the BLOB associated with the locator. (See ["Using Open and Close With LOBs"](#) on page 7-19 for more information.)
- `freeTemporary()`: Frees the storage used by a temporary BLOB. (See ["Working With Temporary LOBs"](#) on page 7-18 for more information.)

- `getBinaryOutputStream()`: Returns a `java.io.OutputStream` to write data to the BLOB as a stream.
- `getBinaryOutputStream(long)`: Returns a `java.io.OutputStream` to write data to the BLOB as a stream. The data is written beginning at the position in the BLOB specified in the argument.
- `getBinaryStream()`: Returns the BLOB data for this `Blob` instance as a stream of bytes.
- `getBinaryStream(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.
- `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 (see `getChunkSize()` below) and is close to 32K.
- `getBytes()`: Reads from the BLOB data, starting at a specified point, into a supplied buffer.
- `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`. (See ["Using Open and Close With LOBs"](#) on page 7-19 for more information.)
- `isTemporary()`: Returns `true` if the BLOB is a temporary BLOB. (See ["Working With Temporary LOBs"](#) on page 7-18 for more information.)
- `length()`: Returns the length of the BLOB in bytes.
- `open()`: Opens the BLOB associated with the locator. (See ["Using Open and Close With LOBs"](#) on page 7-19 for more information.)
- `open(int)`: Opens the BLOB associated with the locator in the mode specified by the argument. (See ["Using Open and Close With LOBs"](#) on page 7-19 for more information.)
- `position()`: Determines the byte position in the BLOB where a given pattern begins.
- `putBytes()`: Writes BLOB data, starting at a specified point, from a supplied buffer.

- `trim(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. (See ["Using Open and Close With LOBs"](#) on page 7-19 for more information.)
- `freeTemporary()`: Frees the storage used by a temporary CLOB. (See ["Working With Temporary LOBs"](#) on page 7-18 for more information.)
- `getAsciiOutputStream()`: Returns a `java.io.OutputStream` to write data to the CLOB as a stream.
- `getAsciiOutputStream(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.
- `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 (see `getChunkSize()` below) and is close to 32K.
- `getCharacterOutputStream()`: Returns a `java.io.Writer` to write data to the CLOB as a stream.
- `getCharacterOutputStream(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.
- `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.
- `isOpen()`: Returns `true` if the CLOB was opened by calling the `open()` method; otherwise, it returns `false`. (See ["Using Open and Close With LOBs"](#) on page 7-19 for more information.)
- `isTemporary()`: Returns `true` if and only if the CLOB is a temporary CLOB. (See ["Working With Temporary LOBs"](#) on page 7-18 for more information.)
- `length()`: Returns the length of the CLOB in characters.
- `open()`: Opens the CLOB associated with the locator. (See ["Using Open and Close With LOBs"](#) on page 7-19 for more information.)
- `open(int)`: Opens the CLOB associated with the locator in the mode specified by the argument. (See ["Using Open and Close With LOBs"](#) on page 7-19 for more information.)
- `position()`: Determines the character position in the CLOB at which a given substring begins.
- `putChars()`: Writes characters from a character array to a specified point in the CLOB data.
- `getSubString()`: Retrieves a substring from a specified point in the CLOB data.
- `putString()`: Writes a string to a specified point in the CLOB data.
- `trim(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 `empty_lob()` method defined in the `oracle.sql.BLOB` and `oracle.sql.CLOB` classes:

- `public static BLOB empty_lob() throws SQLException`

- `public static CLOB empty_lob()` throws `SQLException`

A JDBC driver creates an empty LOB instance without making database round trips. You can use empty LOBs in the following:

- `setXXX()` methods of the `OraclePreparedStatement` class
- `updateXXX()` methods of updatable result sets
- attributes of `STRUCT` objects
- elements of `ARRAY` objects

Note: Because an `empty_lob()` 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.`

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, the space the LOB consumes in temporary table space will not be reclaimed.

You create a temporary LOB with the static method, `createTemporary(Connection, boolean, int)`, defined in 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`, whichever 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, the `isTemporary()` method returns `true`; otherwise, 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.

Note: Failure to free a temporary LOB will result in the storage used by that LOB being unavailable. Frequent failure to free temporary LOBs will result in filling up temporary table space with unavailable LOB storage.

Using Open and Close With LOBs

You do not have to open and close your LOBs. You might choose to open and close them for performance reasons.

If you do not wrap LOB operations inside an Open/Close call operation: 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 operation, 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 opened 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 external binary files (BFILES), using file locators.

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 (or `oracle.jdbc2` interface) for BFILES.
 - If using `getObject()` or `getOracleObject()`, remember to cast the output, as necessary. For more information, see ["Casting Your get Method Return Values"](#) on page 6-10.
-
-

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 an `OracleResultSet`, you can use `getBFILE()` to get the BFILE locator:

```
// 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`, executes 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 (to update a BFILE locator, for example), you can do one of the following:

- Use the standard `setObject()` method.

or:

- 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 (to designate a row number), 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. ["Getting and Passing BFILE Locators"](#) on page 7-20 describes this.

Once you obtain the locator, you can invoke 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. If you want to read and manipulate the data, however, 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 materialized as a Java stream. To read from a BFILE, use the `getBinaryStream()` method of an `oracle.sql.BFILE` object to retrieve the entire 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 cannot use JDBC to create a new BFILE. They 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 below 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. Then execute the statement. 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, then execute the statement. 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, then execute the statement. 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 might 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 an `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. The next section, "[Accessing and Manipulating BFILE Data](#)", describes this.

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 "[Populating a BFILE Column](#)" on page 7-24, getting the locator of the BFILE from row `two` of a table into a result set. The result set is cast to an `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. For a complete BFILE example, see "[BFILES—FileExample.java](#)" on page 20-40.

```
// 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:

- `openFile()`: Opens the external file for read-only access.
- `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.
- `length()`: Returns the length of the BFILE in bytes.
- `position()`: Determines the byte position at which the given byte pattern begins.
- `isFileOpen()`: Determines whether the BFILE is open (for read-only access).

Working with Oracle Object Types

This chapter describes 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 `ORADData` interface. This chapter also describes how JDBC drivers access SQLJ object types in SQL representation.

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](#)
- [SQLJ Object Types](#)

Note: For general information about Oracle object features and functionality, see the *Oracle9i 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 type `Person` that has attributes such as name (type `CHAR`), phone number (type `CHAR`), and employee number (type `NUMBER`).

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. 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* to map to object references, and *custom collection classes* to 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, and 2) populating these classes. You have two options:

- Let JDBC materialize the object as a `STRUCT`. This is described in "[Using the Default STRUCT Class for Oracle Objects](#)" on page 8-3.

or:

- 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. This is described in "[Creating and Using Custom Object Classes for Oracle Objects](#)" on page 8-10.

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. See "[Understanding Type Maps for SQLData Implementations](#)" on page 8-11.

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 are manipulating SQL data. 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 these situations where you don't need the information in a convenient form.

For a complete sample application using the `STRUCT` class to access and manipulate SQL object data, see "[Weakly Typed Objects—PersonObject.java](#)" on page 20-30.

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 (`oracle.jdbc2.Struct` under JDK 1.1.x), and use the following standard methods:

- `getAttributes(map)`: 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 (the default JDBC types).
- `getAttributes()`: This is the same as the preceding `getAttributes(map)` method, except it uses the default type map for the connection.
- `getSQLTypeName()`: Returns a Java `String` that represents the fully qualified name (`schema.sql_type_name`) of the Oracle object type that this `Struct` represents (such as `SCOTT.EMPLOYEE`).

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 (`oracle.jdbc2.Struct` interface under JDK 1.1.x) 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 (`java.sql.Connection`).
- `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—an instance of the `oracle.sql.StructDescriptor` class—to exist for the SQL type (such as `EMPLOYEE`) 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.

`STRUCT` descriptors are further discussed in ["Creating STRUCT Objects and Descriptors"](#) on page 8-4.

Creating STRUCT Objects and Descriptors

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

Steps in Creating StructDescriptor and STRUCT Objects

This section describes how to construct an `oracle.sql.STRUCT` object for a given Oracle object type. To create a `STRUCT` object, you must:

1. Create a `StructDescriptor` object (if one does not already exist) for the given Oracle object type.
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 (SQL structured object). Only one `StructDescriptor` is necessary for each Oracle object type. The driver caches `StructDescriptor` objects to avoid recreating 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, 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:

```
StructDescriptor structdesc = StructDescriptor.createDescriptor  
                                (sql_type_name, connection);
```

Where `sql_type_name` is a Java string containing the name of the Oracle object type (such as `EMPLOYEE`) and `connection` is your connection object.

Once you have your `StructDescriptor` object for the Oracle object type, you can construct the `STRUCT` object. To do this, pass in the `StructDescriptor`, your connection object, and an array of Java objects containing the attributes you want the `STRUCT` to contain.

```
STRUCT struct = new STRUCT(structdesc, connection, attributes);
```

Where `structdesc` is the `StructDescriptor` created previously, `connection` is your connection object, and `attributes` is an array of type `java.lang.Object[]`.

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 typecode, 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()`: Returns the fully qualified SQL type name of the Oracle object (that is, in `schema.sql_type_name` format, such as `CORPORATE.EMPLOYEE`).

- `getLength()`: Returns the number of fields in the object type.
- `getMetaData()`: Returns the meta data regarding this type (like the `getMetaData()` method of a result set object). The returned `ResultSetMetaData` object contains the attribute name, attribute typecode, 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.

The `getMetaData()` method is further discussed in "[Functionality for Getting Object Meta Data](#)" on page 8-49.

Serializable STRUCT Descriptors

As "[Steps in Creating StructDescriptor and STRUCT Objects](#)" on page 8-4 explains, when you create a `STRUCT` object, you first must create a `StructDescriptor` object. Do this by calling the `StructDescriptor.createDescriptor()` method. The `oracle.sql.StructDescriptor` class is serializable, meaning that you can write the complete state of a `StructDescriptor` object to an output stream for later use. Recreate 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—you simply 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.

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 (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, using the type map if it is available, or using default mapping if it is not.

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 `NUMBER` object from column 1 (`col1`) of the table `struct_table`. Because `getObject()` returns an `Object` type, the return is cast to an `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 an
OracleResultSet object 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, referring back to the previous 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` (`oracle.jdbc2.Struct` under JDK 1.1.x). Because `getObject()` returns a `java.lang.Object`, you must cast the output of the method to a `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, use the `getOracleAttributes()` method of the `oracle.sql.STRUCT` class (for a `Struct` instance, you will have to cast to a `STRUCT` instance):

Referring back to the previous examples:

```
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();
```

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 typecode), or cast the statement object to an Oracle statement object 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); //OracleTypes.STRUCT under JDK 1.1.x
```

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 "[ARRAY Automatic Element Buffering](#)" on page 10-9 for a discussion of how to buffer ARRAY elements.)

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 JVM memory without overflow).

Important: 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 a second 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 `getXXX()` and `setXXX()` methods corresponding to the Oracle object's attributes, although this is not necessary. To create and populate the custom classes and provide these read/write capabilities, you can choose between these two 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. If you are using the `SQLData` interface, however, you can only use weak reference types in Java (`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 type `CHAR`) and `EmpNum` (employee number, which is type `NUMBER`). 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 employ 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 "[Using JPublisher to Create Custom Object Classes](#)" on page 8-45 for more information.

Note: If you need to create a custom object class in order to have object-type inheritance, then see "[Object-Type Inheritance](#)" on page 8-29.

The following section compares `ORADATA` and `SQLData` functionality.

Relative Advantages of ORADData versus SQLData

In deciding which of these two interface implementations to use, consider the following:

Advantages of ORADData:

- It does not require an entry in the type map for the Oracle object.
- It has awareness of Oracle extensions.
- You can construct a ORADData 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 (which is in `oracle.sql` format) from the ORADData object, using the `toDatum()` method.
- It provides better performance: ORADData works directly with `Datum` types, which is the internal format used by the driver to hold Oracle objects.

Advantages of SQLData:

- It is a JDBC standard, making your code more portable.

The `SQLData` interface is for mapping SQL objects only. The `ORADData` 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 `ORADData` object from any datatype found in an Oracle database. This could be useful, for example, for serializing RAW data in Java.

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 (SQL 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 `ResultSet` interface `getObject()` method has a signature that lets you specify a type map:

```
rs.getObject(int columnIndex);
```

or:

```
rs.getObject(int columnIndex, Map map);
```

For a description of how to create these custom object classes with `SQLData`, see ["Creating and Using Custom Object Classes for Oracle Objects"](#) on page 8-10.

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. (`ORADData` implementations, by contrast, have their own mapping functionality so that a type map entry is not required. When using a `ORADData` implementation, use the Oracle `getORADData()` 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 (SQL 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 (instances of the `oracle.sql.*` classes) to store attributes.

Creating a Type Map Object 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 as follows:

- under JDK 1.2.x, an instance of a class that implements the standard `java.util.Map` interface
- or:
- under JDK 1.1.x, an instance of a class that extends the standard `java.util.Dictionary` class (or an instance of the `Dictionary` class itself)

You have the option of creating your own class to accomplish this, but under either JDK 1.2.x or JDK 1.1.x, the standard class `java.util.Hashtable` meets the requirement.

Note: If you are migrating from JDK 1.1.x to JDK 1.2.x, 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. Under JDK 1.2.x, both return a `Map` object; under JDK 1.1.x, both return a `Dictionary` object.

The remainder of this section covers the following topics:

- [Adding Entries to an Existing Type Map](#)
- [Creating a New 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 to use any SQL-Java mapping functionality.

Follow these general steps to add entries to an existing type map.

1. Use the `getTypeMap()` method of your `OracleConnection` object to return the connection's type map object. The `getTypeMap()` method returns a `java.util.Map` object (or `java.util.Dictionary` under JDK 1.1.x). 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 type map's `put()` method 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 datatype 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 datatype 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

Follow these 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 more information on the `put()` method, see Step 2 under ["Adding Entries to an Existing Type Map"](#) on page 8-13. 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`, with this statement:

```
newMap.put("CORPORATE.EMPLOYEE", class.forName("Employee"));
```

3. When you finish adding entries to the map, use the `OracleConnection` object's `setTypeMap()` method to overwrite the connection's existing type map. For example:

```
oraconn.setTypeMap(newMap);
```

In this example, `setTypeMap()` overwrites the `oraconn` connection's original map 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 File

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, 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, 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 (`oracle.jdbc2` package under JDK 1.1.x).

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 must be passed in to the `readSQL()` method. `SQLInput` includes a `readXXX()` method for every possible Java type that attributes of an Oracle object might be converted to, such as `readObject()`, `readInt()`, `readLong()`, `readFloat()`, `readBlob()`, and so on. Each `readXXX()` method converts SQL data to Java data and returns it into an output parameter of the corresponding Java type. For example, `readInt()` returns an integer.

The `SQLOutput` implementation is an output stream class, an instance of which must be passed in to the `writeSQL()` method. `SQLOutput` includes a `writeXXX()` method for each of these Java types. Each `writeXXX()` 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 (in other words, 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 datatype that maps to an attribute of the Oracle object, `readSQL()` must call the appropriate `readXXX()` 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, 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 `readXXX()` 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 and populates it with data from a custom object class instance. When the driver calls `writeSQL()`, it passes in this stream parameter.

- For each Java datatype that maps to an attribute of the Oracle object, `writeSQL()` must call the appropriate `writeXXX()` 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 converted by the `writeXXX()` methods to the `SQLOutput` stream so that it can be written to the database once you execute the prepared statement.

"[SQLData Implementation—SQLDataExample.java](#)" on page 20-43 contains a sample implementation of the `SQLData` interface for a given SQL object definition.

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

This section 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()` call triggers `readSQL()` and `readXXX()` calls from the `SQLData` interface, as described above.

Note: If you want to avoid using a 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 an `EmpName` (employee name) of type `CHAR`, and an `EmpNum` (employee number) of type `NUMBER`, then provide a `getEmpName()` method that returns a Java `String` and a `getEmpNum()` method that returns an integer (`int`). Then invoke them in your Java application, as follows:

```
String empname = emp.getEmpName();
int empnumber = emp.getEmpNum();
```

Note: Alternatively, fetch data by using a callable statement object, which also has a `getObject()` method.

Retrieving SQLData Objects from a Callable Statement OUT Parameter

Suppose you have an `OracleCallableStatement` `ocs` that calls a PL/SQL function `GETEMPLOYEE()`. The program passes an employee number (`empnumber`) to the function; the function returns the corresponding `Employee` object.

1. Prepare an `OracleCallableStatement` to call the `GETEMPLOYEE()` function.

```
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 typecode `OracleTypes.STRUCT`. Then, execute the statement.

```
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 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 type `STRUCT`, because the `getObject()` method signature returns the generic `java.lang.Object` type:

```
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, execute the statement.

```
ocs.setObject(1, emp);
ocs.execute();
```

Writing Data to an Oracle Object Using a `SQLData` Implementation

This section 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 datatype object.

```
emp.setEmpName(empname);
emp.setEmpNum(empnumber);
```

This statement uses the `emp` object and the `empname` and `empnumber` variables assigned in ["Reading `SQLData` Objects from a Result Set"](#) on page 8-17.

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 datatype 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 datatype object to the prepared statement.

```
pstmt.setObject(1, emp);
```

4. Execute the statement, which updates the database.

```
pstmt.executeUpdate();
```


Understanding the ORADData 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.ORADData` and `oracle.sql.ORADDataFactory` interfaces (or you can implement `ORADDataFactory` in a separate class). The `ORADData` and `ORADDataFactory` 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 `ORADData` and `ORADDataFactory` interfaces. See ["Using JPublisher to Create Custom Object Classes"](#) on page 8-45.

Understanding ORADData Features

The `ORADData` interface has these advantages:

- It recognizes Oracle extensions to the JDBC; `ORADData` 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: `ORADData` works directly with `Datum` types, the internal format the driver uses to hold Oracle objects.

The `ORADData` and `ORADDataFactory` interfaces do the following:

- The `toDatum()` method of the `ORADData` class transforms the data into an `oracle.sql.*` representation.
- `ORADDataFactory` specifies a `create()` method equivalent to a constructor for your custom object class. It creates and returns a `ORADData` 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 typecode as specified in the `OracleTypes` class.

`ORADData` and `ORADDataFactory` have the following definitions:

```
public interface ORADData
{
    Datum toDatum (java.sql.Connection conn) throws SQLException;
```

```

}

public interface ORADDataFactory
{
    ORADData 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 typecode (from the standard `Types` or `OracleTypes` class) 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 `ORADData`.

To retrieve object data:

- Use the Oracle-specific `OracleResultSet` class `getORADData()` method (assume an `OracleResultSet` object *ors*):

```
ors.getORADData (int col_index, ORADDataFactory factory);
```

This method takes as input the column index of the data in your result set, and a `ORADDataFactory` instance. For example, you can implement a `getORADDataFactory()` method in your custom object class to produce the `ORADDataFactory` instance to input to `getORADData()`. The type map is not required when using Java classes that implement `ORADData`.

or:

- Use the standard `getObject(index, map)` method specified by the `ResultSet` interface to retrieve data as instances of `ORADData`. 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.

To insert object data:

- Use the Oracle-specific `OraclePreparedStatement` class `setORADData()` method (assume an `OraclePreparedStatement` object *ops*):

```
ops.setORADData (int bind_index, ORADData custom_obj);
```

This method takes as input the parameter index of the bind variable and the name of the object containing the variable.

or:

- Use the standard `setObject()` method specified by the `PreparedStatement` interface. You can also use this method, in its different forms, to insert `ORADData` instances without requiring a type map.

The following sections describe the `getORADData()` and `setORADData()` methods.

To continue the example of an Oracle object `EMPLOYEE`, you might have something like the following in your Java application:

```
ORADData datum = ors.getORADData(1, Employee.getORAFactory());
```

In this example, `ors` is an Oracle result set, `getORADData()` is a method in the `OracleResultSet` class used to retrieve a `ORADData` object, and the `EMPLOYEE` is in column 1 of the result set. The static `Employee.getORAFactory()` method will return a `ORADDataFactory` 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:

- `ORADData` and `ORADDataFactory` are defined as separate interfaces so that different Java classes can implement them if you wish (such as an `Employee` class and an `EmployeeFactory` class).
 - To use the `ORADData` interface, your custom object classes must import `oracle.sql.*` (or at least `ORADData`, `ORADDataFactory`, and `Datum`).
-
-

"[ORADData Implementation—ORADDataExample.java](#)" on page 20-47 contains an example implementation of the `ORADData` interface for a given SQL object definition.

Reading and Writing Data with a `ORADData` 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 `ORADData`.

Reading Data from an Oracle Object Using a `ORADData` Implementation

This section summarizes the steps in reading data from an Oracle object into your Java application. These steps apply whether you implement `ORADData` 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 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 connection's default type map 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, 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` (employee number) of type `NUMBER`, provide a `getEmpName()` method that returns a Java string and a `getEmpNum()` method that returns an integer. Then invoke them in your Java application as follows:

```
String empname = emp.getEmpName();  
int empnumber = emp.getEmpNum();
```

Note: Alternatively, you can fetch data into a callable statement object. The `OracleCallableStatement` class also has a `getORAData()` method.

Writing Data to an Oracle Object Using a `ORADData` Implementation

This section summarizes the steps in writing data to an Oracle object from your Java application. These steps apply whether you implement `ORADData` 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 (or had `JPublisher` create it for you).

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 datatype object.

```
emp.setEmpName(empname);  
emp.setEmpNum(empnumber);
```

This statement uses the `emp` object and the `empname` and `empnumber` variables defined in "Reading Data from an Oracle Object Using a `ORADData` Implementation" on page 8-23.

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 datatype object.

```
OraclePreparedStatement opstmt = conn.prepareStatement
    ("UPDATE PERSONNEL SET Employee = ? WHERE Employee.EmpNum = 28959);
```

This assumes `conn` is your `Connection` object.

3. Use the `setORADData()` method of the Oracle prepared statement to bind your Java datatype object to the prepared statement.

```
opstmt.setORADData(1, emp);
```

The `setORADData()` 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 datatype. For example:

```
opstmt.setObject(1, emp);
```

Note: You can use your Java datatype objects as either `IN` or `OUT` bind variables.

Additional Uses for `ORADData`

The `ORADData` 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 (SQL 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 `ORADData` 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 might 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:

- to perform encryption and decryption or validation of data
- to perform logging of values that have been read or are being written
- to parse character columns (such as character fields containing URL information) into smaller components
- to map character strings into numeric constants
- to map data into more desirable Java formats (such as mapping a `DATE` field to `java.util.Date` format)
- to customize data representation (for example, data in a table column is in feet but you want it represented in meters after it is selected)
- to serialize and deserialize Java objects—into or out of `RAW` fields, for example

For example, use `ORADData` to store instances of Java objects that do not correspond to a particular SQL Oracle object type in the database in columns of SQL type `RAW`. The `create()` method in `ORADDataFactory` would have to implement a conversion from an object of type `oracle.sql.RAW` to the desired Java object. The `toDatum()` method in `ORADData` 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 `ORADDataFactory`'s `create()` method 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 `ORADData.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 `ORADData` 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 `ORADData`. For more information on why classes that implement `ORADData` do not need a type map, see ["Understanding the ORADData Interface"](#) on page 8-21.

The Deprecated CustomDatum Interface

As a result of the `oracle.jdbc` interfaces being introduced in Oracle9i as an alternative to the `oracle.jdbc.driver` classes, the `oracle.sql.CustomDatum` and `oracle.sql.CustomDatumFactory` interfaces, formerly used to access customized objects, have been deprecated by the new interfaces—`oracle.sql.ORAData` and `oracle.sql.ORADataFactory`.

The following are the specifications for the `CustomDatum` and `CustomDatumFactory` interfaces:

```
public interface CustomDatum
{
    oracle.sql.Datum toDatum(
        oracle.jdbc.driver.OracleConnection c
    ) throws SQLException ;

    // The following is expected to be present in an
    // implementation:
    //
    // - Definition of public static fields for
    //   _SQL_TYPECODE, _SQL_NAME and _SQL_BASETYPE.
    //   (See Oracle Jdbc documentation for details.)
    //
    // - Definition of
    //   public static CustomDatumFactory
    //   getFactory();
    //
}

public interface CustomDatumFactory
{
    oracle.sql.CustomDatum create(
        oracle.sql.Datum d, int sqlType
    ) throws SQLException;
}
```


Object-Type Inheritance

Object-type inheritance is an Oracle9i feature which allows a new object type to be created by extending another object type. (While Oracle9i does not yet support JDBC 3.0, object-type inheritance is supported and documented.) 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 to do so in addition to any subtype of type T. Oracle9i 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.

Creating Subtypes

Create custom object classes if you want to have Java classes that explicitly correspond to the Oracle object types. (See ["Creating and Using Custom Object Classes for Oracle Objects"](#) on page 8-10.) 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 pass the extended SQL CREATE TYPE command to the `execute()` method of the `java.sql.Statement` interface. For example, to create a type inheritance hierarchy for:

```
PERSON_T
|
STUDENT_T
|
PARTTimestUDENT_T
```

the JDBC code can be:

```
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 an object type. For complete details on the syntax to create subtypes, see the *Oracle9i Application Developer's Guide - Object-Relational Features* for details.

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 `ORADData` or `SQLData` solution in creating classes to map to the hierarchy of object types.

Use of `ORADData` for Type Inheritance Hierarchy

Customized mapping where Java classes implement the `oracle.sql.ORADData` interface is the recommended mapping. (See "[Relative Advantages of `ORADData` versus `SQLData`](#)" on page 8-11.) `ORADData` mapping requires the JDBC application to implement the `ORADData` and `ORADDataFactory` interfaces. The class implementing the `ORADDataFactory` interface contains a factory method that produces objects. Each object represents a database object.

The hierarchy of the class implementing the `ORADData` 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 ORADData Code for the `Person.java` class which implements the `ORADData` and `ORADDataFactory` interfaces:

```
class Person implements ORADData, ORADDataFactory
{
    static final Person _personFactory = new Person();

    public NUMBER ssn;
    public CHAR name;
    public CHAR address;

    public static ORADDataFactory getORADDataFactory()
    {
        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 ORADData 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 ORADDataFactory getORADDataFactory()
    {
        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 above class, `Student`, 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 tabl");
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 `tab1`, such as the following:

```
CREATE TABLE tab1 (idx NUMBER, person PERSON_T);  
INSERT INTO tab1 VALUES (1, PERSON_T (1000, 'Scott', '100 Oracle Parkway'));  
INSERT INTO tab1 VALUES (2, STUDENT_T (1001, 'Peter', '200 Oracle Parkway',  
101, 'CS'));  
INSERT INTO tab1 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 cascade each call to the corresponding methods in the superclass in order 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);
    }
}

```

Customized classes that implement the `SQLData` interface do not have to mirror the database object type hierarchy. For example, you could have declared the above class, `Student`, 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`, `ORADData`, and `ORADDataFactory` interfaces, it is recommended that you use `Oracle9i JPublisher` to automatically generate these classes. The customized classes generated by `JPublisher` that implement the `SQLData`, `ORADData`, and `ORADDataFactory` interfaces, can mirror the inheritance hierarchy.

To learn more about `JPublisher`, see ["Using JPublisher to Create Custom Object Classes"](#) on page 8-45 and the *Oracle9i 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 (`oracle.sql.STRUCT`), `ORADData`, or `SQLData` 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:

```
// tabl.person column can store PERSON_T, STUDENT_T and PARTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tabl");
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.PARTIMESTUDENT_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 (SQL object type).
3. Use the `getObject()` method to access the SQL object values.

The JDBC driver checks the type map for a entry match. If one exists, 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"));  
  
// tabl.person column can store PERSON_T, STUDENT_T and PARTTIMESTUDENT_T  
// objects  
ResultSet rset = stmt.executeQuery ("select person from tabl");  
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 ParttimeStudent");
        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 ORADData Mapping

With ORADData mapping, the JDBC driver returns the database object as an instance of the class implementing the ORADData 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 getObjectORADData(int idx, ORADDataFactory f) method to access database objects. The second parameter of the getObjectORADData() method specifies an instance of the factory class that produces the customized class. The getObjectORADData() 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 first approach avoids the type-map lookup and is therefore more efficient. However, the second approach involves the use of the standard getObject() method. The following code example demonstrates the first approach:

```
// tabl.person column can store both PERSON_T and STUDENT_T objects
```

```
ResultSet rset = stmt.executeQuery ("select person from tabl");
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.println ("This is a ParttimeStudent");
        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 which 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 `ORADData` 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, "Oracle Datatype Classes"](#). 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, "Oracle Datatype Classes"](#). 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:

```
// tabl.person column can store PERSON_T, STUDENT_T and PARTTimestudent_T objects
ResultSet rset = stmt.executeQuery ("select person from tabl");
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

Oracle9iJDBC 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[] getSubtypeNameames()` : 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 isSubType()` : 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 *column* begins with 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 offers a brief overview. For more information, see the *Oracle9i JPublisher User's Guide*.

JPublisher Functionality

You can direct JPublisher to create custom object classes that implement either the `SQLData` interface or the `ORADData` interface, according to how you set the JPublisher type mappings.

If you use the `ORADData` interface, JPublisher will also create a custom reference class to map to object references for the Oracle object type. If you use the `SQLData` interface, 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. The rest of this section lists categories of SQL types and the mapping options available for each category.

For general information about SQL-Java type mappings, see ["Datatype Mappings"](#) on page 3-16.

For more information about JPublisher features or options, see the *Oracle9i JPublisher User's Guide*.

Categories of SQL Types

JPublisher categorizes SQL types into the following groups, with corresponding JPublisher options as noted:

- user-defined types (UDT)—Oracle objects, references, and collections
Use the JPublisher `-usertypes` option to specify the type-mapping implementation for UDTs—either a standard `SQLData` implementation or an Oracle-specific `ORADData` implementation.
- numeric types—anything stored in the database as SQL type `NUMBER`
Use the JPublisher `-numbertypes` option to specify type-mapping for numeric types.
- LOB types—SQL types `BLOB` and `CLOB`
Use the JPublisher `-lobtypes` option to specify type-mapping for LOB types.
- built-in types—anything stored in the database as a SQL type not covered by the preceding categories; for example: `CHAR`, `VARCHAR2`, `LONG`, and `RAW`
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 `ORADData` implementation.
- object-JDBC mapping (for numeric types only) (setting `objectjdbc`)—This is an extension of 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 (for numeric types only) (setting `bigdecimal`)—Uses `java.math.BigDecimal` to map to all numeric attributes; 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` (the default setting) instructs JPublisher to create a `ORADData` implementation for the custom object class.

This will also result in JPublisher producing a `ORADData` 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.

The next section discusses type mapping options that you can use for object attributes.

Note: You can also use JPublisher with a `-usertypes=oracle` setting in creating `ORADData` 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, 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, 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, it will be mapped according to the `-usertypes` setting.

Important: Be especially aware that if you specify a `SQLData` implementation for the custom object class and want the code to be portable, 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 8-1](#) summarizes JPublisher categories for SQL types, the mapping settings relevant for each category, and the default settings.

Table 8-1 *JPublisher SQL Type Categories, Supported Settings, and Defaults*

SQL Type Category	JPublisher Mapping Option	Mapping Settings	Default
UDT types	<code>-usertypes</code>	oracle, jdbc	oracle
numeric types	<code>-numbertypes</code>	oracle, jdbc, objectjdbc, bigdecimal	objectjdbc
LOB types	<code>-lobtypes</code>	oracle, jdbc	oracle
built-in types	<code>-builtintypes</code>	oracle, jdbc	jdbc

Note: The JPublisher `-mapping` option used in previous releases will be deprecated but is currently still supported. For information about how JPublisher converts `-mapping` option settings to settings for the new mapping options, see the *Oracle9i JPublisher User's Guide*.

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.

Functionality for Getting Object Meta Data

The `oracle.sql.StructDescriptor` class, discussed earlier in "[STRUCT Descriptors](#)" on page 8-4 and "[Steps in Creating StructDescriptor and STRUCT Objects](#)" on page 8-4, 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. (Remember that each structured object type must have an associated `StructDescriptor` object.)

The signature of the `StructDescriptor` class `getMetaData()` method is the same as the signature specified for `getMetaData()` in the standard `ResultSet` interface:

- `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 typecode of the specified attribute, according to the `java.sql.Types` and `oracle.jdbc.OracleTypes` classes.
- `String getColumnNameName(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.

As well as the following method, supported only by `StructMetaData`:

- `String getOracleColumnName(int column)`
throws `SQLException`

This 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 `getOracleColumnName()` method, you must cast the `ResultSetMetaData` object (that was returned by the `getMetaData()` method) to a `StructMetaData` object.

Note: In all the preceding method signatures, "column" is something of a misnomer. Where you specify a "column" of 4, 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 as desired—`getColumnName()`, `getColumnType()`, and `getColumnTypeName()`.

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);
        }
    }
}
```

SQLJ Object Types

This section describes how to use Oracle9i JDBC drivers to access SQLJ object types, SQL types for user-defined object types according to the *Information Technology - SQLJ - Part 2: SQL Types using the Java™ Programming Language* document (ANSI NCITS 331.2-2000).

Note: SQLJ object types can either be in serialized or SQL representation. Because Oracle does not support SQLJ object types in serialized representation, this manual describes only SQLJ object types in SQL representation.

According to the *Information Technology - SQLJ - Part 2* document, a *SQLJ object type* is a database object type designed for Java. A SQLJ object type maps to a Java class. Once the mapping is "registered" through the extended SQL `CREATE TYPE` command (a DDL statement), the Java application can insert or select the Java objects directly into or from the database through an Oracle9i JDBC driver. The database SQL engine can access the data fields of these Java objects, stored as SQL attributes in the database, as well as invoke the methods defined in these Java objects.

The extended SQL `CREATE TYPE` command is further discussed in "[Creating a Java Class Definition for a SQLJ Object Type](#)" on page 8-53.

SQLJ object type functionality has the following features:

- Publishes pre-existing Java classes to SQL using the extended SQL `CREATE TYPE` command, creating a mapping between the SQL type and the Java type; no type map is necessary
- Provides a standard way to access Java objects in the database
- Provides a standard way to store Java objects persistently
- Accesses static fields in a Java class using SQL static functions and defines SQL member functions having side effects, which is useful in `UPDATE` statements

Note: SQLJ object type functionality is similar to the use of custom Java classes to map to Oracle object types (SQL object types). The difference between SQLJ object type functionality and custom Java class functionality is that with SQLJ object types, you start with a Java class and then create a corresponding SQL type, instead of the other way around. See "[Creating and Using Custom Object Classes for Oracle Objects](#)" on page 8-10 and "[SQLJ Object Types and Custom Object Types Compared](#)" on page 8-62.

You can obtain additional information on SQLJ object types at the ANSI Web site:

<http://www.ansi.org/>

Creating a SQLJ Object Type in SQL Representation

There are three general steps involved in creating a SQLJ object type in a database:

1. Create the Java class whose instances will be accessed by the database.
See "[Creating a Java Class Definition for a SQLJ Object Type](#)" below.
2. Load the class definition into the database.
See "[Loading the Java Class into the Database](#)" on page 8-55.
3. Use the extended SQL `CREATE TYPE` command in Oracle9i to create a SQLJ object type that represents the Java type.
See "[Creating the SQLJ Object Type in the Database](#)" on page 8-55.

Creating a Java Class Definition for a SQLJ Object Type

To use SQLJ object type functionality, the Java class must implement one of the following Java interfaces:

- `java.sql.SQLData`
- `oracle.sql.ORAData` (and `oracle.sql.ORADataFactory`)

Note: The `ORADData` interface has replaced the `CustomDatum` interface. While the latter interface is deprecated for Oracle9i, it is still supported for backward compatibility. See "[The Deprecated CustomDatum Interface](#)" on page 8-28 for complete details.

The Java class corresponding to a SQLJ object type implements the `SQLData` interface or the `ORADData` and `ORADDataFactory` interfaces, as is the case for custom Java classes that correspond to user-defined Oracle object types in previous Oracle JDBC implementations. The Java class provides methods for moving data between SQL and Java—either using the `readSQL()` and `writeSQL()` methods for classes implementing the `SQLData` interface, or the `toDatum()` method for classes implementing the `ORADData` interface.

The following code shows how the `Person` class for the SQLJ object type, `PERSON_T`, implements the `SQLData` interface:

```
import java.sql.*;
import java.io.*;

public class Person implements SQLData
{
    private String sql_type = "SCOTT.PERSON_T";
    private int ssn;
    private String name;
    private Address address;
    public String getName() {return name;}
    public void setName(String nam) {name = nam;};
    public Address getAddress() {return address;}
    public void setAddress(Address addr) {address = addr;}

    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.readObject();
    }

    public void writeSQL(SQLOutput stream) throws SQLException
```

```

{
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeObject (address);
}

// other methods
public int length () { ... }
}

```

Loading the Java Class into the Database

Once you create the Java class, the next step is to make it available to the database. To do this, use the Oracle `loadjava` tool to load the Java class into the database. See the *Oracle9i Java Tools Reference* for a complete description of the `loadjava` tool.

Note: You can also invoke the `loadjava` tool by calling the `dbms_java.loadjava ('...')` procedure from SQL*Plus, specifying the `loadjava` command line as the input string.

The following command shows the `loadjava` tool loading the `Person` class into the database:

```
% loadjava -u SCOTT/TIGER -r -f -v Person.class
```

Creating the SQLJ Object Type in the Database

The final step in creating a SQLJ object type is to use the extended SQL `CREATE TYPE` command to create the type, specifying the corresponding Java class in the `EXTERNAL NAME` clause.

The follow code shows that `PERSON_T` is the SQLJ object type and `Person` is the corresponding Java class:

```

CREATE TYPE person_t AS OBJECT EXTERNAL NAME 'Person' LANGUAGE JAVA
USING SQLData
    (ss_no number (9) external name 'ssn',
    name VARCHAR2(200) external name 'name',
    address Address_t external name 'address',
    member function length return number external name 'length () return int');
/

```

The extended SQL `CREATE TYPE` command performs the following functions:

- It checks to see if a Java class exists that corresponds to the SQLJ object type and whether this class is public and implements the required interface as specified in the `USING` clause (see the catalog book Magdi told you about).
- It populates the database catalog with the external names for attributes, functions, and the Java class.
- If external attribute names are used, then the extended SQL `CREATE TYPE` command checks for the existence of the Java fields (as specified in the `EXTERNAL NAME` clause) and whether these fields are compatible with corresponding SQL attributes.
- If external attribute names are used, then the extended SQL `CREATE TYPE` command validates the SQL external function against the Java class methods.
- It generates internal classes to support constructors, external static variable names, and external functions that return `self` as a result. The classes are stored in the same schema as the SQLJ object type.

See the *Oracle9i SQL Reference* for a complete description of the extended SQL `CREATE TYPE` command.

Once a SQLJ object type is created, it can be used for the column type of a database table as well as for attributes of other object types. The database SQL engine can access the attributes of the SQLJ object type as well as invoke methods. For example, in SQL*Plus you can do the following:

```
SQL> select col2.ss_no from tab2;
...
SQL> select col2.length() from tab2;
...
```

External Attribute Names The extended SQL `CREATE TYPE` command validates the compatibility between SQLJ object type attributes and corresponding Java fields by comparing the external attribute names (`external name` variables) to the corresponding Java fields. An external attribute name specifies a field in the Java class. For example, in the following code, the `ssn` external name specifies the `ss_no` field in the `Person` Java class:

```
CREATE TYPE person_t AS OBJECT EXTERNAL NAME 'Person' LANGUAGE JAVA
USING SQLData
  (ss_no number (9) external name 'ssn',
  name VARCHAR2(200) external name 'name',
  address Address_t external name 'address',
  member function length return number external name 'length () return int');
```

/

Though optional, external attribute names are good to use when one-to-one correspondences exist between the attributes of a SQLJ object type and the fields of a corresponding Java class. If you choose to use this feature and a declared external attribute name does not exist in the Java class or the SQL attribute is not compatible with the external attribute type, then a SQL error occurs upon executing the extended SQL `CREATE TYPE` command. Or if the provided `SQLData` or `ORADData` interface implementation does not support compatible mapping between a SQL attribute and its corresponding Java field, then an exception may occur.

Note: A SQL attribute declared with an external attribute name may refer to a private Java field.

External SQL Functions The extended SQL `CREATE TYPE` command validates the compatibility between SQLJ object type functions and corresponding Java methods by comparing the external SQL function (`MEMBER FUNCTION` or `STATIC FUNCTION`) to the corresponding Java method. An external SQL function specifies a method in the Java class.

Note: Unlike an external attribute name, an external SQL function is mandatory.

When creating a SQLJ object type in the database, you can declare one or more external SQL functions along with the attributes. [Table 8–2](#) describes the possible kinds of functions that you can use in the creation of a SQLJ object type:

Table 8–2 *Kinds of External SQL Functions for a SQLJ Object Type*

Function	Kind	Syntax
Static functions	Oracle-specific	<code>STATIC FUNCTION foo (...) RETURN NUMBER EXTERNAL NAME 'bar (...) return double'</code>
Member function	SQLJ Part2 Standard	<code>MEMBER FUNCTION foo (...) RETURN NUMBER EXTERNAL NAME 'todo (...) return double'</code>

Table 8–2 Kinds of External SQL Functions for a SQLJ Object Type (Cont.)

Function	Kind	Syntax
Static function that returns the value of a static Java field, which can only be public	Oracle-specific	STATIC FUNCTION foo RETURN NUMBER EXTERNAL VARIABLE NAME 'max_ length'
Static function that calls a constructor in Java	Oracle-specific	STATIC FUNCTION foo (...) RETURN person_t EXTERNAL NAME 'Person (...) return Person'
Member function that has a side effect (changes the state of an object)	SQLJ Part2 Standard	MEMBER FUNCTION foo (...) RETURN SELF AS RESULT EXTERNAL NAME 'dump (...) return Person'

Code Examples The following code shows some typical external SQL functions being declared for a SQLJ object type:

```
CREATE TYPE person_t AS OBJECT EXTERNAL NAME 'Person' LANGUAGE JAVA
USING SQLData
(
    num number external name 'foo',

    STATIC function construct (num number) return person_t
    external name 'Person.Person (int) return Person',
    STATIC function maxvalue return number external variable name 'max_length',
    MEMBER function selfish (num number) return self as result
    external name 'Person.dump (java.lang.Integer) return Person'
)
```

The following code shows how to create the SQLJ object type PERSON_T to represent the Java class Person:

```
CREATE TYPE person_t AS OBJECT EXTERNAL NAME 'Person' LANGUAGE JAVA
USING SQLData
(
    ss_no NUMBER(9) EXTERNAL NAME 'ssn',
    name VARCHAR2(100) EXTERNAL NAME 'name',
    address address_t EXTERNAL NAME 'address',
    MEMBER FUNCTION length RETURN integer EXTERNAL NAME 'length() return int'
);
```

Creating SQLJ Object Types Using JDBC As an alternative to creating a SQLJ object type directly in SQL, using a tool such as SQL*Plus, you can create a SQLJ object type using JDBC code. The following code shows this:

```

Connection conn = ....
Statement stmt = conn.createStatement();
String sql =
    "CREATE TYPE person_t as object external name 'Person' language java
    " using SQLData "+
    "( "+
    "  ss_no number(9), "+
    "  name varchar2(100), "+
    "  address address_t "+
    ")";
stmt.execute(sql);
stmt.close();      // release the resource
conn.close();      // close the database connection

```

Inserting an Instance of a SQLJ Object Type

To create a SQLJ object type instance, the JDBC application creates a corresponding Java instance and then inserts it into the database using the `INSERT` statement. The Java instance can be inserted in one of the following ways:

- A bind variable
- A PL/SQL `IN` parameter
- An object attribute value

Before sending the Java object to the database, the Oracle JDBC driver converts it into a format acceptable to the database SQL engine.

To create a SQLJ object type of `person_t`, as described in previous sections, the JDBC application creates a `Person` object and then inserts it into the database. The following code binds the `person_t` SQLJ object type instance in a SQL insert statement:

```

Person person = new Person();
person.ssn = 1000;
person.name = "SCOTT";
person.address = new Address ("some street", "some city", "CA", 12345);

// insert a SQLJ Object "person_t"
PreparedStatement pstmt = conn.prepareStatement ("insert into tabl (1, ?)");

```

```
pstmt.setObject (1, person);  
pstmt.execute ();
```

Binding a Java instance of a SQLJ object type is equivalent to binding a Java instance of a regular Oracle object type.

Retrieving Instances of a SQLJ Object Type

In a typical JDBC application, Java instances of a SQLJ object type are returned from one of the following:

- Query results
- PL/SQL OUT parameters
- Object type attributes
- Collection elements

In each case, the Oracle JDBC driver materialize the database SQLJ object type instances as instances of the corresponding Java class.

See "[Code Examples](#)" on page 8-58, to learn how the SQLJ object type `person_t` and a database table are created.

Retrieving a SQLJ Object Type Instance Through Database Queries

When a JDBC application queries a column of SQLJ object types in a table, the column values are returned as instances of the Java class that corresponds to the SQLJ object type.

Assume that you have table `tab1` containing column `col1` of SQLJ object type `PERSON_T`. If `PERSON_T` was created to map to the Java class `Person`, then querying `col1` through the Oracle JDBC driver will return the data as instances of the `Person` class. The following code shows this:

```
ResultSet rset = stmt.executeQuery ("select col1 from tab1");  
while (rset.next())  
    Person value = (Person) rset.getObject(1);
```

Notes:

- If the Java class does not exist on the client when the SQLJ object type is returned, a run-time exception occurs.
 - If the Java class exists on the client but has been modified, then the SQLJ object type will only be read or written properly if the `readSQL()` and `writeSQL()` methods for the `SQLData` interface, or the `create()` and `toDatum()` methods for the `ORADData` interface, remain compatible with the original set of SQL attributes.
-
-

Retrieving a SQLJ Object Type Instance as an Output Parameter

Use the `OracleTypes.JAVA_STRUCT` typecode as input to the `registerOutParameter()` method to register a SQLJ object type as a PL/SQL OUT parameter. The following code shows this:

```
CallableStatement cstmt = conn.prepareCall (...);
cstmt.registerOutParameter (1, OracleTypes.JAVA_STRUCT, "SCOTT.PERSON_T");
...
cstmt.execute();
Person value = (Person) cstmt.getObject (1);
```

Meta Data Methods for SQLJ Object Types

Meta data methods are used to query the properties of a datatype. The meta data methods for SQLJ object types are defined in the `oracle.sql.StructDescriptor` class and the `oracle.jdbc.StructMetaData` interface.

To obtain the type descriptor, use the static `createDescriptor()` factory method of the `oracle.sql.StructDescriptor` class as follows:

```
public static StructDescriptor createDescriptor(String name, Connection conn)
    throws SQLException
```

Where `name` is the SQLJ object type and `conn` is the connection to the database.

The `oracle.sql.StructDescriptor` class defines the following meta data (instance) methods:

- `boolean isJavaObject()` : indicates whether the type descriptor points to a SQLJ object type
- `String getJavaClassName()` : returns the name of the Java class corresponding to the SQLJ object type
- `String getLanguage()` : returns the string `JAVA` for a SQLJ object type and returns `null` for an Oracle object type (SQL object type)
- `ResultSetMetaData getMetaData()` : returns the meta data of the SQLJ object type as a result set meta data type (see ["Functionality for Getting Object Meta Data"](#) on page 8-49)
- `getLocalAttributeCount()` : returns the number of local attributes being used, which does not include those used through inheritance

The `oracle.jdbc.StructMetaData` interface provides the following method:

- `String getAttributeJavaName(int idx)`: returns the field name given the relative position of the SQL attribute; the relative position starts at zero and inherited attributes are included

SQLJ Object Types and Custom Object Types Compared

This section describes the differences between SQLJ object types and Oracle object types (custom object types).

Table 8–3 SQLJ Object Type and Custom Object Type Features Compared

Feature	SQLJ Object Type Behavior	Custom Object Type Behavior
Typecodes	Use the <code>OracleTypes.JAVA_STRUCT</code> typecode to register a SQLJ object type as a SQL OUT parameter. The <code>OracleTypes.JAVA_STRUCT</code> typecode is also used in the <code>_SQL_TYPECODE</code> field of a class implementing the <code>ORADData</code> or <code>SQLData</code> interface. This typecode is reported in a <code>ResultSetMetaData</code> instance and meta data or stored procedure.	Use the <code>OracleTypes.STRUCT</code> typecode to register a custom object type as a SQL OUT parameter. The <code>OracleTypes.STRUCT</code> typecode is also used in the <code>_SQL_TYPECODE</code> field of a class implementing the <code>ORADData</code> or <code>SQLData</code> interface. (See "ORADData Implementation—ORADDataExample.java" on page 20-47 for a code example.) The <code>OracleTypes.STRUCT</code> typecode is reported in a <code>ResultSetMetaData</code> instance and meta data or stored procedure.
Creation	Create a Java class implementing the <code>SQLData</code> or <code>ORADData</code> and <code>ORADDataFactory</code> interfaces first and then load the Java class into the database. Next, issue the extended SQL <code>CREATE TYPE</code> command to create the SQLJ object type.	Issue the extended SQL <code>CREATE TYPE</code> command for a custom object type and then create the <code>SQLData</code> or <code>ORADData</code> Java wrapper class using <code>JPublisher</code> , or do this manually. See "Using JPublisher to Create Custom Object Classes" on page 8-45 for complete details.
Method Support	Supports external names, constructor calls, and calls for member functions with side effects. See Table 8–2, "Kinds of External SQL Functions for a SQLJ Object Type" on page 8-57 for a complete description.	There is no default class for implementing type methods as Java methods. Some methods may also be implemented in SQL.
Type Mapping	Type mapping is automatically done by the extended SQL <code>CREATE TYPE</code> command. However, the SQLJ object type must have a defining Java class on the client.	Register the correspondence between SQL and Java in a type map. Otherwise, the type is materialized as <code>oracle.sql.STRUCT</code> .
Corresponding Java Class is Missing	If the corresponding Java class is missing when a SQLJ object type is returned to the client, you will receive an exception.	If the corresponding Java class is missing when a custom object type is returned to the client, then <code>oracle.sql.STRUCT</code> is used.
Inheritance	There are rules for mapping SQL hierarchy to a Java class hierarchy. See the <i>Oracle9i SQL Reference</i> for a complete description of these rules.	There are no mapping rules.

Working with Oracle Object References

This chapter describes Oracle extensions to standard 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 (pointers) to Oracle database objects. Oracle JDBC provides support for object references as:

- columns in a SELECT-list
- IN or OUT bind variables
- attributes in an Oracle object
- elements in a collection (array) 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* in this manual and must implement the `oracle.sql.ORADData` interface.

The `oracle.sql.REF` class implements the standard `java.sql.Ref` interface (`oracle.jdbc2.Ref` under JDK 1.1.x).

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 (for example, `EMPLOYEE`).

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 runtime.

For more information about custom reference classes, see ["Custom Reference Classes with JPublisher"](#) on page 9-10.

For a complete sample application using the `REF` class to access SQL object data, see ["Weakly Typed Object References—StudentRef.java"](#) on page 20-33.

Notes:

- If you are using the `oracle.sql.ORAData` interface for custom object classes, you will presumably use `ORAData` for corresponding custom reference classes as well. If you are using the standard `java.sql.SQLData` interface for custom object classes, however, you can only use weak Java types for references (`java.sql.Ref` or `oracle.sql.REF`). The `SQLData` interface is for mapping SQL object types only.
 - You cannot create `REF` objects in your JDBC application; you can only retrieve existing `REF` objects from the database.
 - 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 summarizes the following:

- statement and result set getter and setter methods for passing REF instances from and to the database
- REF class functionality to get and set object attributes

Remember that you can use custom reference classes instead of the `ARRAY` class. See ["Custom Reference Classes with JPublisher"](#) on page 9-10.

Object Reference Getter and Setter Methods

Use the following result set, callable statement, and prepared statement methods to retrieve and pass object references. Code examples are provided later in the chapter.

Result Set and Callable Statement Getter Methods The `OracleResultSet` and `OracleCallableStatement` classes support `getREF()` and `getRef()` methods to retrieve REF objects as output parameters—either as `oracle.sql.REF` instances or `java.sql.Ref` instances (`oracle.jdbc2.Ref` under JDK 1.1.x). 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, respectively, an `oracle.sql.REF` instance or a `java.sql.Ref` instance (`oracle.jdbc2.Ref` under JDK 1.1.x).

Key REF Class Methods

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 (for example, `EMPLOYEE`).

This is a standard method specified by the `java.sql.Ref` interface.

- `getValue()`: Retrieves the referenced object from the database, allowing you to access its attribute values. It optionally takes a type map object, or else 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 as input (either a `STRUCT` instance or an instance of a custom object class).

This method is an Oracle extension.

Retrieving and Passing an Object Reference

This section discusses JDBC functionality for retrieving and passing object references.

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 a Java `Address` object.

Here is the code for these steps (other than adding `Address` to the type map), where `stmt` is a previously defined statement object. The `PEOPLE` database table is defined earlier in this section:

```
ResultSet rs = stmt.executeQuery("SELECT col3 FROM PEOPLE");
while (rs.next())
{
```

```

REF ref = ((OracleResultSet)rs).getREF(1);
Address a = (Address)ref.getValue();
}

```

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()` allows 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 an `OracleCallableStatement`:

```
OracleCallableStatement ocs =
    (OracleCallableStatement)conn.prepareCall("{? = call func()}");
```

2. Register the OUT parameter with this form of the `registerOutParameter()` method:

```
ocs.registerOutParameter
    (int param_index, int sql_type, String sql_type_name);
```

Where *param_index* is the parameter index and *sql_type* is the SQL typecode (in this case, `OracleTypes.REF`). 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 (as in ["Retrieving and Passing an Object Reference"](#) on page 9-6), then ADDRESS is the *sql_type_name* that should be passed in.

3. Execute the call:

```
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.

Continuing the example in "[Retrieving and Passing an Object Reference](#)" on page 9-6, 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 (if one does not already exist) that corresponds to the database object.

For example, you can use the code in the section ["Retrieving and Passing an Object Reference"](#) on page 9-6 to retrieve the reference to a database ADDRESS object:

```
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 (this example omits the content for the constructor of the `Address` class) that corresponds to the database ADDRESS object. Use the `setValue()` method of the REF class to set the value of the database object:

```
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 described under "[Creating and Using Custom Object Classes for Oracle Objects](#)" on page 8-10. 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 JPublisher's `-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 runtime—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. (Specifying the `SQLData` implementation is accomplished by setting JPublisher's UDT attributes mapping option to `jdbc`.)

For more information about JPublisher, see "[Using JPublisher to Create Custom Object Classes](#)" on page 8-45, or refer to the *Oracle9i JPublisher User's Guide*.

Working with Oracle Collections

This chapter describes Oracle extensions to standard 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 \(Arrays\)](#)
- [Overview of Collection \(Array\) Functionality](#)
- [Creating and Using Arrays](#)
- [Using a Type Map to Map Array Elements](#)
- [Custom Collection Classes with JPublisher](#)

Oracle Extensions for Collections (Arrays)

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, although "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 lets you use arrays as any of the following:

- columns in a SELECT-list
- IN or OUT bind variables
- attributes in an Oracle object
- as elements of other arrays (Oracle9i only)

The rest of this section discusses creating and materializing collections.

The remainder of the chapter describes how to access and update collection data through Java arrays. For a complete code example of creating a table with a collection column and then manipulating and printing the contents, see "[Weakly Typed Arrays—ArrayExample.java](#)" on page 20-35.

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* in this manual. 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 (`oracle.jdbc2.Array` under JDK 1.1.x).

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. You cannot write to the array, however, as there are no setter methods.

Custom collection classes, as with the `ARRAY` class, allow 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 runtime.

Furthermore, custom collection classes produced by `JPublisher` offer the feature of being writable, with individually accessible elements. (This is also something you could implement in a custom collection class yourself.)

Note: There is no difference in your 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.

For more information about custom collection classes, see "[Custom Collection Classes with JPublisher](#)" on page 10-27.

Creating Collections

This section presents background information about creating Oracle 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, as in the following SQL syntax:

```
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 datatype. Each element has an index, which is a number corresponding to the element's position 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 datatype. 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, it can also be viewed as a multi-column table, with a column for each attribute of the object type. Create a nested table with this SQL syntax:

```
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 Multi-Level Collection Types

The most common way to create a new multi-level 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` using the `execute()` method:

```
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 multi-level collection types have been created, they can be used as both columns of a base table as well as attributes of a object type.

See the *Oracle9i Application Developer's Guide - Object-Relational Features* for the SQL syntax to create multi-level collections types and how to specify the storage tables for inner collections.

Note: Multi-level collection types are available only for Oracle9i.

Overview of Collection (Array) 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 (`oracle.jdbc2.Array` interface under JDK 1.1.x), provides the necessary functionality to access and update the data of an Oracle collection (either a VARRAY or nested table).

This section discusses the following:

- statement and result set getter and setter methods for passing collections to and from the database as Java arrays
- ARRAY descriptors and ARRAY class methods

Remember that you can use custom collection classes instead of the `ARRAY` class. See "[Custom Collection Classes with JPublisher](#)" on page 10-27.

Array Getter and Setter Methods

Use the following result set, callable statement, and prepared statement methods to retrieve and pass collections as Java arrays. Code examples are provided later in the chapter.

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 (`oracle.jdbc2.Array` under JDK 1.1.x). You can also use the `getObject()` method. These methods take as input a `String` column name or `int` column index.

Prepared and Callable Statement Setter Methods The `OraclePreparedStatement` and `OracleCallableStatement` classes support `setARRAY()` and `setArray()` methods to take updated ARRAY objects as bind variables and pass them to the database. You can also use the `setObject()` method. These methods take as input a `String` parameter name or `int` parameter index as well as, respectively, an `oracle.sql.ARRAY` instance or a `java.sql.Array` instance (`oracle.jdbc2.Array` under JDK 1.1.x).

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—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.

`ARRAY` descriptors are further discussed in ["Creating ARRAY Objects and Descriptors"](#) on page 10-11.

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()`: Identical to `getArray()`, but retrieves the elements in `oracle.sql.*` format.
- `getBaseType()`: Returns the SQL typecode for the array elements (see ["Class oracle.jdbc.OracleTypes"](#) on page 5-22 for information about typecodes).
- `getBaseTypeName()`: Returns the SQL type name of the elements of this array.
- `getSQLTypeName()` (Oracle extension): Returns the fully qualified SQL type name of the array as a whole.
- `getResultSet()`: Materializes the array elements as a result set.
- `getJavaSQLConnection()`: Returns the connection instance (`java.sql.Connection`) 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. (See "[STRUCT Automatic Attribute Buffering](#)" on page 8-9 for a discussion of how to buffer STRUCT attributes.)

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 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, 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.

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 (if one does not already exist) for the array.
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 recreating 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, when you create a collection with the `CREATE TYPE` statement:

```
CREATE TYPE num_varray AS varray(22) OF NUMBER(5,2);
```

Where `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 go to the database to gather meta data) to the constructor.

```
ArrayDescriptor arraydesc = ArrayDescriptor.createDescriptor
    (sql_type_name, connection);
```

Where `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);
```

Where `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 Multi-Level Collections

As with single-level collections, the JDBC application can create an `oracle.sql.ARRAY` instance to represent a multi-level 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 multi-level collection type. The second argument is the current database connection. And the third argument is a `java.lang.Object` that holds the multi-level collection elements. This is the same constructor used to create single-level collections, but in Oracle9i, this constructor is enhanced to create multi-level 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 multi-level 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 above 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 typecode of the array's elements, and, if it is an array of structured objects, the SQL name of the elements. The descriptor also contains the information on about to convert 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 an element's typecode and type name:

- `createDescriptor()`: This is a factory for `ArrayDescriptor` instances; looks up the name in the database and determine the characteristics of the array.
- `getBaseType()`: Returns the integer typecode associated with this `ARRAY` descriptor (according to integer constants defined in the `OracleTypes` class, which "[Package oracle.jdbc](#)" on page 5-16 describes).
- `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 (`java.sql.Connection`) that was used in creating the ARRAY descriptor (a new descriptor must be created for each connection instance).

Note: In releases prior to Oracle9i, you cannot use a collection within a collection. You can, however, use a structured object with a collection attribute, or a collection with structured object elements. In Oracle9i, you can use a collection within a collection.

Serializable ARRAY Descriptors

As "[Steps in Creating ArrayDescriptor and ARRAY Objects](#)" on page 10-11 discusses, 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. Recreate 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, 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.

Retrieving the Array

You can retrieve a SQL array from a result set by casting the result set to an `OracleResultSet` object 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 an `oracle.sql.ARRAY` object.

Data Retrieval Methods

Once you have the array in 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 (`java.sql.Array` under JDK 1.2.x or `oracle.jdbc2.Array` under JDK 1.1.x). The `getOracleArray()` method retrieves the element values of the array into a `Datum[]` array. The elements are of the `oracle.sql.*` datatype 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 `getXXX()` method. In contrast, `getArray()` returns the entire contents of the nested table at one time.

The `getResultSet()` method uses the connection's default type map to determine the mapping between the SQL type of the Oracle object and its corresponding Java datatype. If you do not want to use the connection's default type map, 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 into a `java.lang.Object` instance that you can cast as appropriate (see "[Comparing the Data Retrieval Methods](#)" on page 10-17). 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, the use by that method of `oracle.sql.Datum` instances avoids the expense of data conversion from SQL to Java. The data inside a `Datum` (or subclass) instance remains in raw SQL format.

If you use `getResultSet()` to return an array of primitive datatypes, 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; the second column stores the element value.

If you use `getArray()` to retrieve an array of primitive datatypes, then a `java.lang.Object` that contains the element values is returned. The elements of this array are of the Java type corresponding to the SQL type of the elements. For example:

```
BigDecimal[] values = (BigDecimal[]) intArray.getArray();
```

Where `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` datatype 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 datatypes according to the default mapping. This is because these methods use the connection's default type map 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.

For more information on using type maps with arrays, see ["Using a Type Map to Map Array Elements"](#) on page 10-25.

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 described above, 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 datatype of the elements of the original array. For example:

```
Datum arraydata[] = arr.getOracleArray();
```

Where `arr` is an `oracle.sql.ARRAY` object. For an example of retrieving an array and its contents, see ["Weakly Typed Arrays—ArrayExample.java"](#) on page 20-35.

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 an `OracleResultSet` object; `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 `my_array` are of the SQL datatype `NUMBER`, it must first be cast to the `BigDecimal` datatype. 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 typecode " + 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, you would first get the result set object, 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 Multi-Level Collection Elements

The `oracle.sql.ARRAY` class provides three methods (which can be overloaded) to access collection elements. For Oracle9i JDBC drivers, these methods are extended to support multi-level collections. The three methods are the following:

- `getArray()` method : JDBC standard
- `getOracleArray()` method : Oracle extension
- `getResultSet()` method : JDBC standard

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 multi-level 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 multi-level collection elements. For multi-level 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 (use similar steps to pass an array to a callable statement). Note that 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 (unless one has already been created for this SQL type). See ["Steps in Creating ArrayDescriptor and ARRAY Objects"](#) on page 10-11 for information about creating `ArrayDescriptor` objects.

```
ArrayDescriptor descriptor = ArrayDescriptor.createDescriptor
    (sql_type_name, connection);
```

Where `sql_type_name` is a Java string specifying the user-defined SQL type name of the array, and `connection` is your `Connection` object. See ["Oracle](#)

[Extensions for Collections \(Arrays\)](#)" on page 10-2 for information about SQL typenames.

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);
```

Where *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 execute.
4. Cast your prepared statement to an `OraclePreparedStatement` and use the `setARRAY()` method of the `OraclePreparedStatement` object to pass the array to the prepared statement.

```
(OraclePreparedStatement)stmt.setARRAY(parameterIndex, array);
```

Where *parameterIndex* is the parameter index, and *array* is the `oracle.sql.ARRAY` object you constructed previously.

5. Execute the prepared statement.

Passing an Array to a Callable Statement

To retrieve a collection as an `OUT` parameter in PL/SQL blocks, execute the following to register the bind type for your `OUT` parameter.

1. Cast your callable statement to an `OracleCallableStatement`:

```
OracleCallableStatement ocs =  
    (OracleCallableStatement)conn.prepareCall("{? = call func()}");
```

2. Register the `OUT` parameter with this form of the `registerOutParameter()` method:

```
ocs.registerOutParameter  
    (int param_index, int sql_type, string sql_type_name);
```

Where *param_index* is the parameter index, *sql_type* is the SQL typecode, and *sql_type_name* is the name of the array type. In this case, the *sql_type* is `OracleTypes.ARRAY`.

3. Execute the call:

```
ocs.execute();
```

4. Get the value:

```
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. For instructions on how to add entries to an existing type map or how to create a new type map, see ["Understanding Type Maps for SQLData Implementations"](#) on page 8-11.

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 example below).

The `EmployeeObj` custom object class in this example implements the `SQLData` interface. ["Custom Object Class—SQLData Implementation"](#) on page 20-44 contains the code that creates the `EmployeeObj` type.

```
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, invoke 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 (it is also possible to implement such functionality yourself):

- They are strongly typed. This can help you find coding errors during compilation that might not otherwise be discovered until runtime.
- 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. (This is also something you could implement in a custom collection class yourself.)

A custom collection class must satisfy three requirements:

- It must implement the `oracle.sql.ORAData` interface described under ["Creating and Using Custom Object Classes for Oracle Objects"](#) on page 8-10. 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 (this is the case with a JPublisher-produced custom collection class).

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 `ORADData` implementation. This will be true if JPublisher's `-usertypes` mapping option is set to `oracle`, which is the default.

You cannot use a `SQLData` implementation for a custom collection class (that implementation is for custom object classes only). 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 runtime—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.

For more information about JPublisher, see ["Using JPublisher to Create Custom Object Classes"](#) on page 8-45, or refer to the *Oracle9i JPublisher User's Guide*.

Result Set Enhancements

Standard JDBC 2.0 features in JDK 1.2.x include enhancements to result set functionality—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](#)

The Oracle JDBC drivers also include extensions to support these features in a JDK 1.1.x environment.

For more general and conceptual information about JDBC 2.0 result set enhancements, refer to the Sun Microsystems JDBC 2.0 API specification.

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.

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.

This section provides an overview of these enhancements, types, and categories.

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 allows you to move a specified number of rows forward or backward from the current row. Absolute positioning allows you to move to a specified row number, counting from either the beginning or the end of the result set.

Under JDBC 1.0 (in JDK 1.1.x) you can scroll only forward, using the `next()` method as described in "[Process the Result Set](#)" on page 3-11, and there is no positioning functionality. You can start only at the beginning and iterate row-by-row until the end.

Under JDBC 2.0 (in JDK 1.2.x), 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.

Sensitivity is not an option in a JDBC 1.0/non-scrollable result set.

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, 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 three result set types are available with JDBC 2.0:

- forward-only (JDBC 1.0 functionality—not scrollable, not positionable, and not sensitive)
- scroll-sensitive (scrollable and positionable; also sensitive to underlying database changes)
- scroll-insensitive (scrollable and positionable but not sensitive to underlying database changes)

Note: The sensitivity of a scroll-sensitive result set (how often it is updated to see external changes) is affected by fetch size. See [Fetch Size](#) on page 11-24 and "[Oracle Implementation of Scroll-Sensitive Result Sets](#)" on page 11-30.

Updatability

Updatability refers to the ability to update data in a result set and then (presumably) 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, but not under JDBC 1.0.

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 (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*:

- 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 ROWIDs.

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 to fail. *Do not specify scrollability for a large result set.*

Scrollable cursors in the Oracle server, and therefore a server-side cache, will be supported in a future Oracle release.

Oracle JDBC Implementation for Result Set Updatability

To support updatability, Oracle JDBC uses ROWIDs 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, 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 executing a query. The result set produced by the query will then use your specified caching mechanism.

Creating Scrollable or Updatable Result Sets

Under JDBC 1.0, no special attention is required in creating and using a result set. A result set is produced automatically to store the results of a query, and no result set types or categories must be specified, because there is only one kind of result set available—forward-only/read-only. For example (given a connection object `conn`):

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");
```

In using JDBC 2.0 result set enhancements, however, you may specify the result set type (for scrollability and sensitivity) and the concurrency type (for updatability) when you create a generic statement or prepare a prepared statement or callable statement that will execute a query.

(Note, however, that callable statements are intended to execute stored procedures and functions and rarely return a result set. Still, the callable statement class is a subclass of the prepared statement class and so inherits this functionality.)

This section discusses the creation of result sets to use JDBC 2.0 enhancements.

Specifying Result Set Scrollability and Updatability

Under JDBC 2.0, Connection classes have `createStatement()`, `prepareStatement()`, and `prepareCall()` method signatures 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`

Note: See ["Oracle Implementation of Scroll-Sensitive Result Sets"](#) on page 11-30 for information about possible performance impact.

And you can specify one of the following static constant values for concurrency type:

- `ResultSet.CONCUR_READ_ONLY`
- `ResultSet.CONCUR_UPDATABLE`

Note: If you are using the Oracle JDBC drivers in a JDK 1.1.x environment, the static constants discussed here are part of the Oracle extensions, belonging only to the `OracleResultSet` class, which you must specify. For example:

```
OracleResultSet.TYPE_SCROLL_SENSITIVE
```

instead of:

```
ResultSet.TYPE_SCROLL_SENSITIVE
```

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 Following is an example of a prepared statement object that specifies a scroll-sensitive and updatable result set for queries executed through that statement (where `conn` is a connection object):

```
...
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 execute, 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 executed, 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, or call the methods described in ["Verifying Result Set Type and Concurrency Type"](#) on page 11-12.

FOR UPDATE Clause Limitation in an Updatable Result Set

A query cannot have the `FOR UPDATE` clause in the `SELECT` statement if you are using an updatable result set. If you use the `FOR UPDATE` clause and try to update a result set, an `SQLException` will be thrown.

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 *"`. (But see the workaround below.)
- 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.
- A query cannot use `ORDER BY`.

To produce a scroll-sensitive result set:

- A query cannot use `"SELECT *"`. (But see the workaround below.)
- A query can select from only a single table.
- A query cannot use `ORDER BY`.

In fact, you cannot use `ORDER BY` for any result set where you will want to refetch rows. This applies to scroll-insensitive/updatable result sets as well as scroll-sensitive result sets. (See "[Summary of New Methods for Result Set Enhancements](#)" on page 11-32 for general information about refetching.)

Workaround As a workaround for the "`SELECT *`" limitation, you can use table aliases as in the following example:

```
SELECT t.* FROM TABLE t ...
```

Hint: 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. (You can try this out using SQL*Plus, for example.)

Result Set Downgrade Rules

If the specified result set type or concurrency type is not feasible, 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`.

Furthermore:

- 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:

- Criteria that would prevent the JDBC driver from fulfilling the result set type specifications are listed in "[Result Set Limitations](#)" on page 11-10.
 - 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 executed, 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 (result set type `TYPE_SCROLL_SENSITIVE` or `TYPE_SCROLL_INSENSITIVE`) allow 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.

For a complete sample application demonstrating this functionality, see "[Positioning in a Result Set—ResultSet2.java](#)" on page 20-52.

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.

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`
- `void afterLast()` throws `SQLException`
- `boolean first()` throws `SQLException`
- `boolean last()` throws `SQLException`
- `boolean absolute(int row)` throws `SQLException`
- `boolean relative(int row)` throws `SQLException`

Note: You cannot position a forward-only result set. Any attempt to position it or to determine the current position will result in a SQL exception.

beforeFirst() Method 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.

afterLast() Method 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.

first() Method Positions to the first row of the result set, or returns `false` if there are no rows in the result set.

last() Method Positions to the last row of the result set, or returns `false` if there are no rows in the result set.

absolute() Method Positions to an absolute row from either the beginning or end of the result set. If you input a positive number, it positions from the beginning; if you input a negative number, 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()`.

relative() Method Moves to a position relative to the current row, either 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.

Important: 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 SQL exception.

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` (and 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 (has passed the first row).

Backward versus Forward Processing

You can process the entire result set going forward, using the `next()` method as in JDBC 1.0. This is documented in "[Process the Result Set](#)" on page 3-11. 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 (where `conn` is a connection object):

```
...
/* 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 (and typically do) 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 SQL exception.

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` allows 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.

A `DELETE` operation in a result set, however, is immediately executed (but not necessarily committed) in the database as well.

For sample applications demonstrating this functionality, see ["Inserting and Deleting Rows in a Result Set—ResultSet3.java"](#) on page 20-55 and ["Updating Rows in a Result Set—ResultSet4.java"](#) on page 20-59.

Note: When using an updatable result set, it is typical to also make it scrollable. This allows 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.

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`

Important: 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 executed 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 executed 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 (presuming a result set `rs`):

```
...  
rs.absolute(5);  
rs.deleteRow();  
...
```

See "[Positioning in a Scrollable Result Set](#)" on page 11-13 for information about the positioning methods.

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.

Refer to "[Seeing Internal Changes](#)" on page 11-27 for more information.

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.

See "[Positioning in a Scrollable Result Set](#)" on page 11-13 for information about the positioning methods.

Here are the steps for updating a row in the result set and database:

1. Call the appropriate `updateXXX()` methods to update the data in the columns you want to change.

With JDBC 2.0, a result set object has an `updateXXX()` method for each datatype, as with the `setXXX()` 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 datatype 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 executed 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 executed operation is committed immediately.

If you choose to cancel the changes before copying them to the database, 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 (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 `getXXX()` 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).

Refer to "[Seeing Internal Changes](#)" on page 11-27 for more information.

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 will be 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 executing 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 `updateXXX()` methods to write data to the columns. For example:

```

rs.updateString(1, "mystring");
rs.updateFloat(2, 10000.0f);

```

(Note that 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 SQL exception.

It is permissible, however, 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 executed 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 (neither scroll-sensitive, scroll-insensitive, nor forward-only) can see a row inserted by a result set `INSERT` operation.

Refer to "[Seeing Internal Changes](#)" on page 11-27 for more information.

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...
...
```

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 still valid when a driver tries to send an `UPDATE` or `DELETE` operation to the database, the operation will be executed.

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 executing 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 executes a query, it receives the result set 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 (see ["Oracle Row Prefetching"](#) on page 12-20 for more information).

JDBC 2.0 also allows 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 executed through that statement object.

Fetch size is also used in a result set. When the statement object executes 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. (Also note that changes made to a statement object's fetch size 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. See ["Refetching Rows"](#) on page 11-26.)

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 executing the query. If you set the fetch size to N, then N rows are fetched with each trip to the database.

After you have executed 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. (See "[Refetching Rows](#)" on page 11-26.)

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 "[Oracle Row Prefetching](#)" on page 12-20 for a general discussion of this Oracle feature.

Note: Do not mix the JDBC 2.0 fetch size 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 (described above in "[Fetch Size](#)" on page 11-24). 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 `refreshRow()` method signature:

- `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

Oracle JDBC might support additional result set categories in future releases.

For a code sample that explicitly refetches data using the `refreshRow()` method, see "[Refetching Rows in a Result Set—ResultSet6.java](#)" on page 20-64.

Note: Scroll-sensitive result set functionality is implemented through implicit calls to `refreshRow()`. See "[Oracle Implementation of Scroll-Sensitive Result Sets](#)" on page 11-30 for details.

Seeing Database Changes Made Internally and Externally

This section discusses the ability of a result set to see the following:

- its own changes (DELETE, UPDATE, or INSERT operations within the result set), referred to as *internal changes*
- changes made from elsewhere (either from your own transaction outside the result set, or from other committed transactions), referred to as *external changes*

Near the end of the section is a summary table.

Note: External changes are referred to as "other's changes" in the Sun Microsystems JDBC 2.0 specification.

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 (UPDATE, DELETE, or INSERT). This is discussed at various points throughout the "[Updating Result Sets](#)" section beginning on page 11-18, and is summarized as follows:

- Internal DELETE operations are visible for scrollable result sets (scroll-sensitive or scroll-insensitive), 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 (forward-only, scroll-sensitive, or scroll-insensitive).
- Internal INSERT operations are never visible, regardless of the result set type (neither forward-only, scroll-sensitive, nor scroll-insensitive).

An internal change being "visible" essentially means that a subsequent `getXXX()` call will see the data changed by a preceding `updateXXX()` call on the same data item.

JDBC 2.0 DatabaseMetaData objects include the following methods to verify this. Each takes a result set type as input (`ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_SENSITIVE`, or `ResultSet.TYPE_SCROLL_INSENSITIVE`).

- `boolean ownDeletesAreVisible(int)` throws `SQLException`
- `boolean ownUpdatesAreVisible(int)` throws `SQLException`

- `boolean ownInsertsAreVisible(int)` throws `SQLException`

Note: When you make an internal change that causes a trigger to execute, the trigger changes are effectively external changes. However, if the trigger affects data in the row you are updating, 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.

For implementation details of scroll-sensitive result sets, including exactly how and how soon external updates become visible, see "[Oracle Implementation of Scroll-Sensitive Result Sets](#)" on page 11-30.

JDBC 2.0 `DatabaseMetaData` objects include the following methods to verify this. Each takes a result set type as input (`ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_SENSITIVE`, or `ResultSet.TYPE_SCROLL_INSENSITIVE`).

- `boolean othersDeletesAreVisible(int)` throws `SQLException`
- `boolean othersUpdatesAreVisible(int)` throws `SQLException`
- `boolean othersInsertsAreVisible(int)` throws `SQLException`

Note: Explicit use of the `refreshRow()` method, described in "[Refetching Rows](#)" on page 11-26, 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. Each takes a result set type as input (`ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_SENSITIVE`, or `ResultSet.TYPE_SCROLL_INSENSITIVE`).

- `boolean deletesAreDetected(int)` throws `SQLException`
- `boolean updatesAreDetected(int)` throws `SQLException`
- `boolean insertsAreDetected(int)` throws `SQLException`

It follows, then, that result set methods specified by JDBC 2.0 to detect changes—`rowDeleted()`, `rowUpdated()`, and `rowInserted()`—will always return `false` with the Oracle JDBC drivers. There is no use in calling them.

Summary of Visibility of Internal and External Changes

[Table 11-1](#) 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 11-1 *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

For implementation details of scroll-sensitive result sets, including exactly how and how soon external updates become visible, see "[Oracle Implementation of Scroll-Sensitive Result Sets](#)" on page 11-30.

Notes:

- Remember that explicit use of the `refreshRow()` method, described in "[Refetching Rows](#)" on page 11-26, is distinct from the concept of "visibility" of external changes. This is discussed in "[Seeing External Changes](#)" on page 11-28.
 - 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`. This is further discussed in "[Visibility versus Detection of External Changes](#)" on page 11-29.
-
-

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 (as described in "[Positioning in a Scrollable Result Set](#)" on page 11-13), 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 (see ["Fetch Size"](#) on page 11-24). 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 (described in ["Refetching Rows"](#) on page 11-26), 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.

For a sample application that demonstrates the functionality of a scroll-sensitive result set, see ["Scroll-Sensitive Result Set—ResultSet5.java"](#) on page 20-61.

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 tradeoff 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. These methods are more fully discussed throughout this chapter.

Modified Connection Methods

Following is an alphabetical 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 now allows 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 now allows 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 now allows you to specify result set type and concurrency type when you create a `CallableStatement` object.

New Result Set Methods

Following is an alphabetical 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 (you will not be at a valid current row after this call).
- `void beforeFirst() throws SQLException`
Move to before the first row in the result set (you will not be at a valid current row after this call).

- `void cancelRowUpdates()` throws `SQLException`
Cancel an `UPDATE` operation on the current row. (Call this after the `updateXXX()` 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 (either `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`).
- `int getFetchSize()` throws `SQLException`
Check the fetch size to determine how many rows are fetched in each database round trip (also available in statement objects).
- `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 (either `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_SENSITIVE`, or `ResultSet.TYPE_SCROLL_INSENSITIVE`).
- `void insertRow()` throws `SQLException`
Write a result set `INSERT` operation to the database. Call this after calling `updateXXX()` 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 (also available in statement objects).

- `void updateRow()` throws `SQLException`

Write an `UPDATE` operation to the database after using `updateXXX()` methods to update the data values.

- `void updateXXX()` throws `SQLException`

Set or update data values in a row to be updated or inserted. There is an `updateXXX()` method for each datatype. After calling all the appropriate `updateXXX()` methods for the columns to be updated or inserted, call `updateRow()` for an `UPDATE` operation or `insertRow()` for an `INSERT` operation.

New Statement Methods

Following is an alphabetical summary of new 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 (also available in result set objects).
- `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 (also available in result set objects).
- `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).

New Database Meta Data Methods

Following is an alphabetical summary of new 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`.

Performance Extensions

This chapter describes the Oracle performance extensions to the JDBC standard. In the course of discussing update batching, it also includes a discussion of the standard update-batching model provided with JDBC 2.0.

This chapter covers the following topics:

- [Update Batching](#)
- [Additional Oracle Performance Extensions](#)

Note: For a general overview of Oracle extensions and detailed discussion of Oracle packages and type extensions, see [Chapter 5, "Overview of Oracle 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 in this manual as *update batching* and in the Sun Microsystems JDBC 2.0 specification 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 Sun Microsystems JDBC 2.0 Specification, which is referred to as *standard update batching*
- the Oracle-specific model, independent of the Sun Microsystems 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 the syntax of one model or the other, but not both. The Oracle JDBC driver will throw exceptions when you mix these syntaxes.

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 batch (accumulate) for each trip to the database. As soon as that many operations have been added to the batch, the batch is executed. Note the following:

- You can set a default batch for the connection object, which applies to any prepared statement executed 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 execute 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 execute 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, here is how to choose between the two models:

- 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* Oracle prepared statement objects. In an Oracle 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.

Note that because Oracle update batching is vendor-specific, you must actually use (or cast to) `OraclePreparedStatement` objects, not general `PreparedStatement` objects.

- To adhere to the JDBC 2.0 standard, Oracle's implementation of standard update batching supports callable statements and generic statements, as well as prepared statements. You can migrate standard update batching syntax into an Oracle JDBC application without difficulty.
- You can batch *only* `UPDATE`, `INSERT`, or `DELETE` operations. Executing 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.

Note that with standard update batching, you can use either standard `PreparedStatement`, `CallableStatement`, and `Statement` objects, or Oracle-specific `OraclePreparedStatement`, `OracleCallableStatement`, and `OracleStatement` objects.

Oracle Update Batching

The Oracle update batching feature associates a batch value (limit) with each prepared statement object. With Oracle update batching, instead of the JDBC driver executing a prepared statement each time its `executeUpdate()` method is called, the driver adds the statement to a batch of accumulated execution requests. The driver will pass all the operations to the database for execution 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 allows you to set a default batch value for the Oracle connection as a whole, and this batch value is relevant to any Oracle prepared statement in the connection. For any particular Oracle prepared statement, a method in the `OraclePreparedStatement` class allows you to set a statement batch value that overrides the connection batch value. You can also override both batch values by choosing to manually execute the pending batch.

Notes:

- Do not mix standard update batching syntax with Oracle update batching syntax in the same application. The JDBC driver will throw an exception when you mix these syntaxes.
 - Disable auto-commit mode if you use either update batching model. In case an error occurs while you are executing a batch, this allows you the option of committing or rolling back the operations that executed 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 (default) 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 (or becomes) a stream type, then the Oracle JDBC driver sets the batch value to 1 and sends any queued requests to the database for execution.
- The Oracle JDBC driver automatically executes the `sendBatch()` method of an Oracle prepared statement in any of the following circumstances: 1) the connection receives a `COMMIT` request, either as a result of invoking the `commit()` method or as a result of auto-commit mode; 2) the statement receives a `close()` request; or 3) the connection receives a `close()` request.

Note: A connection `COMMIT` request, statement close, or connection close has no effect on a pending batch if you use standard update batching—only if you use Oracle update batching.

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. See ["Specifying a Database URL and Properties Object"](#) on page 3-6.

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 executes.

1. Write your prepared statement and specify input values for the first row:

```
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 an `OraclePreparedStatement` object, 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:

```
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 execute 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 could accomplish this by casting your connection to an `OracleConnection` object 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 and specify input values for the first row as usual, then execute the statement:

```
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 executed 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, the data will still not be sent to the database, because the batch value in effect for the statement is the connection batch value: 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 execute 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 executed, but also issues an implicit `sendBatch()` call to execute 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 executed, 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: Oracle Update Batching

The following example illustrates how you use the Oracle update batching feature. It assumes you have imported the `oracle.driver.*` interfaces.

```
...
Connection conn =
    DriverManager.getConnection("jdbc:oracle:oci8:", "scott", "tiger");

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();
...
```

For complete sample applications for Oracle update batching, including how to execute the batch both implicitly and explicitly, see ["Oracle Update Batching with Implicit Execution—SetExecuteBatch.java"](#) on page 20-71 and ["Oracle Update Batching with Explicit Execution—SendBatch.java"](#) on page 20-73.

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 Sun Microsystems JDBC 2.0 Specification. Because it is a JDBC 2.0 feature, it is intended for use in a JDK 1.2.x environment. To use standard update batching in a JDK 1.1.x environment, you must cast to Oracle statement objects.

This model, unlike the Oracle update batching model, depends on explicitly adding statements to the batch using an `addBatch()` method and explicitly executing the

batch using an `executeBatch()` method. (In the Oracle model, you invoke `executeUpdate()` as in a non-batching situation, but whether an operation is added to the batch or the whole batch is executed is typically determined implicitly, depending on whether a pre-determined batch value is reached.)

Notes:

- Do not mix standard update batching syntax with Oracle update batching syntax in the same application. The Oracle JDBC driver will throw exceptions when these syntaxes are mixed.
 - Disable auto-commit mode if you use either update batching model. In case an error occurs while you are executing a batch, this allows you the option of committing or rolling back the operations that executed successfully prior to the error.
-
-

Limitations in the Oracle Implementation of Standard Batching

Note the following limitations and implementation details regarding Oracle's implementation of standard update batching:

- In Oracle JDBC applications, update batching is intended for use with prepared statements that are being executed 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 syntax for `Statement` and `CallableStatement` objects, you are unlikely to see performance improvement.
- Oracle's implementation of standard update batching does not support stream types as bind values. (This is also true of Oracle update batching.) 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 interfaces `oracle.jdbc.OracleStatement`, `OraclePreparedStatement`, and `OracleCallableStatement`, respectively.

For a `Statement` object (or `OracleStatement`), the `addBatch()` method takes a Java string with a SQL operation as input. For example (assume a `Connection` instance `conn`):

```
...
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.

(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 executions 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 `setXXX()` 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 (again assuming a `Connection` instance `conn`):

```
...
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 executions of a single prepared statement, as in this example.

Executing the Batch

To execute 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 executes 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 executing the batch and sometimes containing information about the number of rows affected. This is discussed in "[Update Counts in the Oracle Implementation of Standard Batching](#)" on page 12-15.

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 executed, operations are performed in the order in which they were batched.
 - The statement batch is reset to empty once `executeBatch()` has returned.
 - An `executeBatch()` call closes the statement object's current result set, if one exists.
-
-

Committing the Changes in the Oracle Implementation of Standard Batching

After you execute the batch, you must still commit the changes, presuming auto-commit is disabled as recommended.

Calling `commit()` commits non-batched operations and commits batched operations for statement batches that have been executed, but for the Oracle implementation of standard batching, has no effect on pending statement batches that have *not* been executed.

Clearing the Batch

To clear the current batch of operations instead of executing 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 executed successfully (no batch exception is thrown), 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 or callable 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 because Oracle JDBC cannot use true batching for generic and callable statements in the Oracle implementation of standard update batching.

In your code, upon successful execution of a batch, you should be prepared to handle either -2's or true update counts in the array elements. For a successful batch execution, the array contains either all -2's or all positive integers.

Note: For information about possible values in the update counts array for an *unsuccessful* batch execution, see ["Error Handling in the Oracle Implementation of Standard Batching"](#) on page 12-16.

Example: Standard Update Batching

This example combines the sample fragments in the previous sections, accomplishing the following steps:

- disabling auto-commit mode (which you should always do when using either update batching model)
- creating a prepared statement object
- adding operations to the batch associated with the prepared statement object
- executing the batch
- committing the operations from the batch

Assume a `Connection` instance `conn`:

```
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 executed successfully. This is discussed in the next section ("[Error Handling in the Oracle Implementation of Standard Batching](#)").

For a complete sample application, see "[Standard Update Batching—BatchUpdates.java](#)" on page 20-69.

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 execution stops and a `java.sql.BatchUpdateException` is generated (a subclass of `java.sql.SQLException`).

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 exception:

- 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 execution of a batch, you should be prepared to handle either -3's or true update counts in the array elements when an exception occurs. For a failed batch execution, you will have either a full array of -3's or a partial array of positive integers.

Intermixing Batched Statements and Non-Batched Statements

You cannot call `executeUpdate()` for regular, non-batched execution of an operation if the statement object has a pending batch of operations (essentially, if the batch associated with that statement object is non-empty).

You can, however, intermix batched operations and non-batched operations in a single statement object if you execute non-batched operations either prior to adding any operations to the statement batch or after executing 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 scout = 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 scout = 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 executed batches, but will not affect any pending batches.

Premature Batch Flush

Premature batch flush happens due to a change in cached meta data. Cached meta data 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 shown below:

```

HashTable info = new HashTable ();
info.put ("user", "SCOTT");
info.put ("passwd", "TIGER");
// other properties
...

// property: batch flush type
info.put ("AccumulateBatchResult", "false");

Connection con = DriverManager.getConnection ("jdbc:oracle:oci8:@", info);

```

Note: The `AccumulateBatchResult` property is set to `true` by default, in Oracle9i.

Example:

```

((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, discussed previously, 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. For more information, see "[Specifying a Database URL and Properties Object](#)" on page 3-6.

Oracle Row Prefetching

Oracle JDBC drivers include extensions that allow 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.

Note: With JDBC 2.0, the ability to preset the fetch size has become standard functionality. For information about the standard implementation of this feature, see "[Fetch Size](#)" on page 11-24.

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 (any type of 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 an `OracleStatement`, `OraclePreparedStatement`, or `OracleCallableStatement` object, as applicable, if it is not already one of these.
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, use the `getRowPrefetch()` method of the Statement object, 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 an `OracleConnection` object.
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. See "[Specifying a Database URL and Properties Object](#)" on page 3-6.

Notes:

- Do not mix the JDBC 2.0 fetch size 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: 1) explicitly refetching rows in a result set through the result set `refreshRow()` method available with JDBC 2.0 (relevant for scroll-sensitive/read-only, scroll-sensitive/updatable, and scroll-insensitive/updatable result sets); and 2) the "window" size of a scroll-sensitive result set, affecting how often automatic refetches are performed. The Oracle row-prefetch value will be overridden, however, by any setting of the fetch size. See "[Fetch Size](#)" on page 11-24 for more information.
-

Example: Row Prefetching The following example illustrates the row-prefetching feature. It assumes you have imported the `oracle.jdbc.*` interfaces.

```
Connection conn =
    DriverManager.getConnection("jdbc:oracle:oci8:", "scott", "tiger");

//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();
```

For complete sample applications, including how to set the connection default row-prefetch value and the statement row-prefetch value, see ["Oracle Row Prefetching Specified in Connection—RowPrefetch_connection.java"](#) on page 20-74 and ["Oracle Row Prefetching Specified in Statement—RowPrefetch_statement.java"](#) on page 20-76.

Oracle Row-Prefetching Limitations

There is no maximum prefetch setting, but empirical evidence suggests that 10 is effective. Oracle does not recommend exceeding this value in most situations. If you do not set the default row-prefetch value for a connection, 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 connection's default row-prefetch setting have no effect on the statement's row-prefetch setting.

If a column of a result set is of datatype `LONG` or `LONG RAW` (that is, the streaming types), JDBC changes the statement's row-prefetch setting to 1, even if you never actually read a value of either of those types.

If you use the form of the `DriverManager` class `getConnection()` method that takes a `Properties` object as an argument, then you can set the connection's default row-prefetch value that way. See ["Specifying a Database URL and Properties Object"](#) on page 3-6.

Defining Column Types

Oracle JDBC drivers enable you to inform the driver of the types of the columns in an upcoming query, saving a round trip to the database that would otherwise be necessary to describe the table.

When standard JDBC performs a query, it first uses a round trip to the database to determine the types that it should use for the columns of the result set. Then, when JDBC receives data from the query, it converts the data, as necessary, as it populates the result set.

When you specify column types for a query, you avoid the first round trip to the database. The server, which is optimized to do so, performs any necessary type conversions.

For a complete sample application, see "[Oracle Column Type Definitions—DefineColumnType.java](#)" on page 20-78.

Following these general steps to define column types for a query:

1. Cast your statement object to an `OracleStatement`, `OraclePreparedStatement`, or `OracleCallableStatement` object, as applicable, if it is not already one of these.
2. If necessary, use the `clearDefines()` method of your `Statement` object to clear any previous column definitions for this `Statement` object.
3. For each column of the expected result set, invoke the `defineColumnType()` method of your `Statement` object, passing it these parameters:

- column index (integer)
- typecode (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`). Typecodes for standard types are identical in these two classes.

- type name (string) (structured objects, object references, and arrays only)

For structured objects, object references, and arrays, you must also specify the type name (for example, `Employee`, `EmployeeRef`, or `EmployeeArray`).

- (optionally) 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.

For example, assuming `stmt` is an Oracle statement, use this syntax:

```
stmt.defineColumnType(column_index, typeCode);
```

or (recommended if the column is `VARCHAR` or equivalent and you know the length limit):


```
stmt.defineColumnType(column_index, typeCode, max_size);
```

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 maximum field size set in `defineColumnType()`

or:

- the maximum field size set in `setMaxFieldSize()`

or:

- the natural maximum size of the datatype

Once you complete these steps, use the statement's `executeQuery()` method to perform the query.

Note: You must define the datatype for *every* column of the expected result set. If the number of columns for which you specify types does not match the number of columns in the result set, the process fails with a SQL exception.

Example: Defining Column Types The following example illustrates the use of this feature. It assumes you have imported the `oracle.jdbc.*` interfaces.

```
Connection conn =
    DriverManager.getConnection("jdbc:oracle:oci8:", "scott", "tiger");

Statement stmt = conn.createStatement();

/*Ask for the column as a string:
 *Avoid a round trip to get the column type.
 *Convert from number to string on the server.
 */
((OracleStatement)stmt).defineColumnType(1, Types.VARCHAR);

ResultSet rset = stmt.executeQuery("select empno from emp");
```

```

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

stmt.close();

```

As this example shows, you must cast the statement (`stmt`) to type `OracleStatement` in the invocation of the `defineColumnType()` method. The connection's `createStatement()` method 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` typecode.

[Table 12-1](#) lists the valid column definition arguments you can use in the `defineColumnType()` method.

Table 12-1 Valid Column Type Specifications

If the column has Oracle SQL type:	You can use <code>defineColumnType()</code> 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

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. See "[Specifying a Database URL and Properties Object](#)" on page 3-6.

If you are using a standard `java.sql.Connection` object, you must cast it to `OracleConnection` to use `setRemarksReporting()`.

Example: 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 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 (the `catalog` parameter) and output (the `catalog` column in the returned `ResultSet`). On input, the construct `" "` (the 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 stand-alone and packaged objects (same 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 `" "` (the 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 (same 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 "%".

Statement Caching

This chapter describes the benefits and use of statement caching, an Oracle JDBC extension.

The following topics are discussed:

- [About Statement Caching](#)
- [Using Statement Caching](#)

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.

Statement caching can:

- Prevent the overhead of repeated cursor creation
- Prevent repeated statement parsing and creation

Basics of Statement Caching

Use a statement cache to cache statements associated with a particular physical connection. Enabling statement caching automatically enables implicit and explicit statement caching. Both these types of statement caching share the same cache.

For a simple connection, the cache is associated with an `OracleConnection` object. For a pooled connection, the cache is associated with an `OraclePooledConnection` object. The `OracleConnection` and `OraclePooledConnection` objects include methods to enable statement caching. When you enable statement caching, a statement object is cached when you call the "close" methods.

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 pooled connection, all the logical connections will use the *same* cache. If you try to enable statement caching on a logical connection of a pooled connection, this will throw an exception.

As mentioned previously, there are two forms of statement caching:

- Implicit
- Explicit

Implicit Statement Caching

When you enable statement caching, *implicit statement caching* 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.

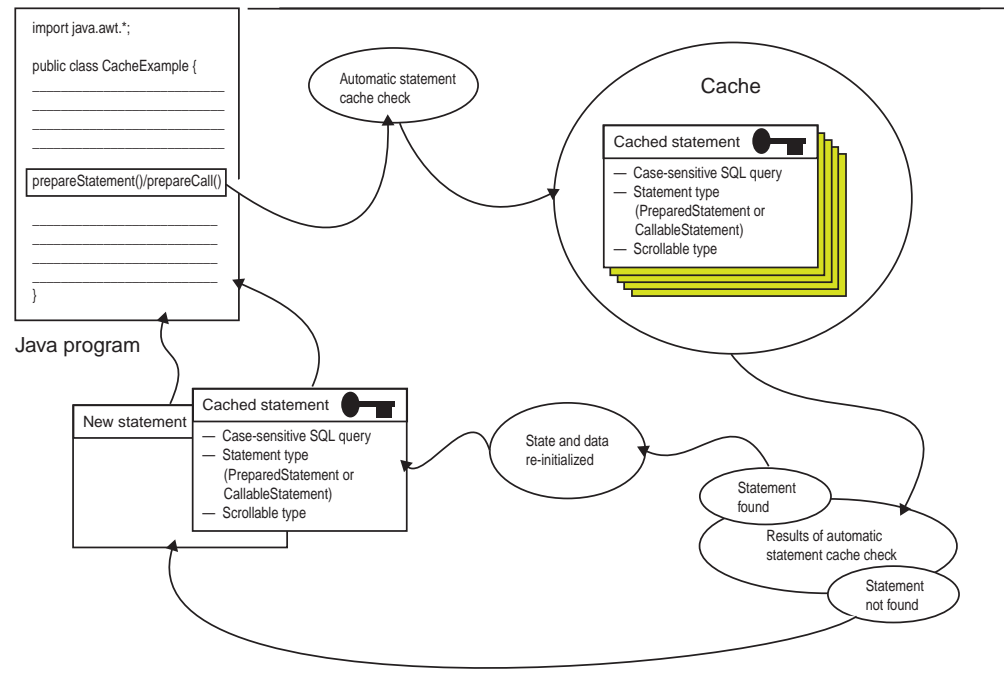
Plain statements are not implicitly cached because implicit statement caching uses a SQL string as a key, and plain statements are created without a SQL string. Therefore, implicit statement caching applies only to the

OraclePreparedStatement and OracleCallableStatement objects, which are created with a SQL string. When one of these statements is created, 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 (case-sensitive) to one in the cache.
- The statement type must be the same (prepared or callable).
- The scrollable type of result sets produced by the statement must be the same (forward-only or scrollable). You can determine the scrollability when you create the statement. (See "[Specifying Result Set Scrollability and Updatability](#)" on page 11-8 for complete details.)

Figure 13-1 illustrates the implicit statement caching concept.

Figure 13-1 Implicit Statement Caching Process



If a match is found during the cache search, the cached statement is returned. If a match is not found, then a new statement is created and returned. The new 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. The Least Recently Used (LRU) scheme performs the statement cache operation.

Note: By default, the JDBC driver does not clear metadata. However, if you call the `setStmtCacheSize()` method with the `clearMetaData` parameter set to `true`, then metadata is also cleared. Clearing metadata is not typical. See ["Enabling and Disabling Statement Caching"](#) on page 13-6 for complete details.

You can prevent a particular statement from being implicitly cached by calling a special method. This method is discussed in ["Disabling Implicit Statement Caching for a Particular Statement"](#) on page 13-8.

Explicit Statement Caching

When you enable statement caching, *explicit statement caching*, which is based on a user-defined key, enables you to cache and retrieve selected prepared, callable, and plain statements. The *key* is an arbitrary Java string that you provide.

To explicitly cache a statement and to retrieve an explicitly cached statement, use the Oracle "WithKey" methods. Employing these methods, specify the key as an input parameter. As the name implies, explicit statement caching is performed on each statement object you want to cache.

If the JDBC driver finds the matching statement with the specified key, then the statement is returned. If the JDBC driver cannot find a matching statement in the cache, it will return a `null` value.

Comparing Implicit and Explicit Statement Caching

Because explicit statement caching retains the statement data and state, in addition to the metadata, it has a performance edge over implicit statement caching, which retains only the metadata. However, because explicit statement caching saves all three types of information for re-use, you must be cautious when using this type of

caching—you may not be aware of what was retained for data and state in the previous statement.

Implicit statement caching uses standard methods to allocate and retrieve statement objects. Explicit statement caching uses standard methods to allocate statements and specialized Oracle "WithKey" methods to cache and retrieve statement objects.

Implicit statement caching uses the SQL string of a prepared or callable statement as the key, requiring no action on your part. Explicit statement caching requires you to provide a Java string, which it uses as the key.

During implicit statement caching, if the JDBC driver cannot find a statement in cache, it will automatically create one. During explicit statement caching, if the JDBC driver cannot find a matching statement in cache, it will return a `null` value.

[Table 13-1](#) compares the different methods employed in implicit and explicit statement caching.

Table 13-1 Comparing Methods Used in Statement Caching

	Allocate	Cache	Retrieve
Implicit	<code>prepareStatement()</code> <code>prepareCall()</code>	<code>close()</code>	<code>prepareStatement()</code> <code>prepareCall()</code>
Explicit	<code>createStatement()</code> <code>prepareStatement()</code> <code>prepareCall()</code>	<code>closeWithKey()</code>	<code>createStatementWithKey()</code> <code>prepareCallWithKey()</code> <code>prepareStatementWithKey()</code>

Using Statement Caching

This section discusses the following topics:

- [Enabling and Disabling Statement Caching](#)
- [Checking for Statement Creation Status](#)
- [Physically Closing a Cached Statement](#)
- [Using Implicit Statement Caching](#)
- [Using Explicit Statement Caching](#)

Enabling and Disabling Statement Caching

You can dynamically enable and disable statement caching. Enable statement caching by calling the `setStmtCacheSize()` method of your connection object, setting the cache to a value greater than zero. The cache size you specify is the maximum number of statements in the cache.

The `setStmtCacheSize()` method has the following signatures:

- `setStmtCacheSize (int size)`
- `setStmtCacheSize (int size, boolean clearMetaData)`

The `size` parameter sets the maximum number of statements in the cache. The optional `clearMetaData` parameter allows you to specify if metadata is to be cleared when the cached statement is retrieved.

Notes:

- When you enable or disable statement caching for a particular physical connection, you do this for *both* implicit and explicit statement caching. 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. Therefore, consider this when you set the statement cache size
-
-

The following code specifies a cache size of ten statements:

```
((OracleConnection)conn).setStmtCacheSize(10);
```

Disable statement caching by calling the `setStmtCacheSize()` method and setting the cache to zero. To check the cache size, use the `getStmtCacheSize()` method.

The following code disables statement caching by setting the cache size to zero. Additionally, it verifies the new cache setting in the last line of code.

```
((OracleConnection)conn).setStmtCacheSize(0);
System.out.println("Stmt Cache size is " +
    ((OracleConnection)conn).getStmtCacheSize());
```

Checking for Statement Creation Status

By calling the `creationState()` method of a statement object, you can determine if a statement was newly created or if it was retrieved from cache on an implicit or explicit lookup. The `creationState()` method returns the following integer values for plain, prepared, and callable statements:

- `NEW` - The statement was newly created.
- `IMPLICIT` - The statement was retrieved on an implicit statement lookup.
- `EXPLICIT` - The statement was retrieved on an explicit statement lookup.

For example, the JDBC driver returns `OracleStatement.EXPLICIT` for an explicitly cached statement. The following code checks the statement creation status for `stmt`:

```
int state = ((OracleStatement)stmt).creationState()
    ...(process state)
```

Physically Closing a Cached Statement

With statement caching enabled, you cannot truly 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 three conditions: (1) when the associated connection is closed, (2) when the cache reaches its size limit and the least recently used statement object is preempted from cache by the LRU scheme, or (3) if you call the `close()` method on a statement for which statement caching is disabled. (See "[Disabling Implicit Statement Caching for a Particular Statement](#)" on page 13-8 for more details.)

Using Implicit Statement Caching

Once you enable statement caching, by default all prepared and callable statements are automatically cached. Implicit statement caching includes the following steps:

1. Enable statement caching as described in ["Enabling and Disabling Statement Caching"](#) on page 13-6.
2. Allocate a statement using one of the standard methods.
3. Optionally disable implicit statement caching for any particular statement you do not want to cache.
4. Cache the statement using the `close()` method.
5. Retrieve the implicitly cached statement by calling the appropriate standard "prepare" method.

For a complete code example of implicit statement caching, see ["Implicit Statement Caching—StmtCache1.java"](#) on page 20-79. The following sections explain the implicit statement caching steps in more detail.

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. (These are methods of the connection object.)

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 statement caching enabled for a connection, by default all callable and prepared statements of that connection are automatically cached. To prevent a particular callable or prepared statement from being implicitly cached, use the `setDisableStmtCaching()` method of the statement object. To help you manage cache space, you can call the `setDisableStmtCaching()` method on any infrequently used statement.

The following code disables implicit statement caching for `pstmt`:

```
PreparedStatement pstmt = conn.prepareStatement ("SELECT 1 from DUAL");  
((OraclePreparedStatement)pstmt).setDisableStmtCaching (true);  
pstmt.close ();
```

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 recall an implicitly cached statement, call either the `prepareStatement()` or `prepareCall()` method, depending on the statement type.

The following code recalls `pstmt` from cache using the `prepareStatement()` method:

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

If you call the `creationState()` method on the `pstmt` statement object, the method returns `IMPLICIT`. If the `pstmt` statement object was not in cache, then the `creationState()` method returns `NEW` to indicate a new statement was recently created by the JDBC driver.

[Table 13–2](#) describes the methods used to allocate statements and retrieve implicitly cached statements.

Table 13–2 *Methods Used in Statement Allocation and Implicit Statement Caching*

Method	Functionality for Implicit Statement Caching
<code>prepareStatement()</code>	triggers a cache search that either finds and returns the desired cached <code>OraclePreparedStatement</code> object or allocates a new <code>OraclePreparedStatement</code> object if a match is not found
<code>prepareCall()</code>	triggers a cache search that either finds and returns the desired cached <code>OracleCallableStatement</code> object or allocates a new <code>OracleCallableStatement</code> object if a match is not found

Using Explicit Statement Caching

A plain, prepared, or callable statement can be explicitly cached when you enable statement caching. Explicit statement caching includes the following steps:

1. Enable statement caching as described in ["Enabling and Disabling Statement Caching"](#) on page 13-6.
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.

For a complete code example of explicit statement caching, see ["Explicit Statement Caching—StmtCache2.java"](#) on page 20-82. The following sections explain the explicit statement caching steps in more detail.

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. (These are methods of the connection object.)

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 `createStatementWithKey()`, `prepareStatementWithKey()`, or `prepareCallWithKey()` method, depending on the statement type.

If you retrieve a statement with a specified key, the JDBC driver searches the cache for the statement, based on the specified key. If a match is found, the matching statement is returned, along with its state, data, and metadata. This information is returned as it was when last closed. If a match is not found, the JDBC driver returns null.

The following code recalls `pstmt` from cache using the "mykey" key with the `prepareStatementWithKey()` method. Recall that the `pstmt` statement object was cached with the "mykey" key.

```
pstmt = ((OracleConnection)conn).prepareStatementWithKey ("mykey");
```

If you call the `creationState()` method on the `pstmt` statement object, the method returns `EXPLICIT`.

Important: When you retrieve an explicitly cached statement, be sure to 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 `prepareStatementWithKey()` method to retrieve that statement from cache. The JDBC driver cannot verify the type of statement it is returning.

Table 13–3 describes the methods used to retrieve explicitly cached statements.

Table 13–3 *Methods Used to Retrieve Explicitly Cached Statements*

Method	Functionality for Explicit Statement Caching
<code>createStatementWithKey()</code>	specifies the key needed to retrieve a plain statement from cache
<code>prepareStatementWithKey()</code>	specifies the key needed to retrieve a prepared statement from cache
<code>prepareCallWithKey()</code>	specifies the key needed to retrieve a callable statement from cache

Connection Pooling and Caching

This chapter covers the Oracle JDBC implementations of (1) data sources, a standard facility for specifying resources to use, including databases; (2) connection pooling, which is a framework for caches of database connections; and (3) connection caching, including documentation of a sample Oracle implementation. You will also find related discussion of Oracle JDBC support for the standard Java Naming and Directory Interface (JNDI).

The following topics, which apply to all Oracle JDBC drivers, are described in this chapter:

- [Data Sources](#)
- [Connection Pooling](#)
- [Connection Caching](#)

Notes: This chapter describes features of the Sun Microsystems JDBC 2.0 Standard Extension API, which are available through the `javax` packages from Sun Microsystems. These packages are not part of the standard JDK, but relevant packages are included with the `classes111.zip` and `classes12.zip` files.

For further information on listed topics, refer to the Sun Microsystems specification for the JDBC 2.0 Standard Extension API. For information about additional connection pooling functionality specific to the OCI driver, see "[OCI Driver Connection Pooling](#)" on page 16-2.

Data Sources

The JDBC 2.0 extension API introduces the concept of *data sources*, which are standard, general-use objects for specifying databases or other resources to use. Data sources can optionally be bound to Java Naming and Directory Interface (JNDI) entities so that you can access databases by logical names, for convenience and portability.

This functionality is a more standard and versatile alternative to the connection functionality described under "[Open a Connection to a Database](#)" on page 3-3. The data source facility provides a complete replacement for the previous JDBC `DriverManager` facility.

You can use both facilities in the same application, but ultimately developers will be encouraged to use data sources for their connections, regardless of whether connection pooling or distributed transactions are required. Eventually, Sun Microsystems will probably deprecate `DriverManager` and related classes and functionality.

For further introductory and general information about data sources and JNDI, refer to the Sun Microsystems specification for the JDBC 2.0 Optional Package.

A Brief Overview of Oracle Data Source Support for JNDI

The standard Java Naming and Directory Interface, or JNDI, provides a way for applications to find and access remote services and resources. These services can be any enterprise services, but for a JDBC application 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 Oracle9i CD, but is not included in the `classes12.zip` and `classes111.zip` files. 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 that has been tested with the Oracle drivers.)

Data Source Features and Properties

"[First Steps in JDBC](#)" on page 3-2 includes sections on how to use the JDBC `DriverManager` class to register driver classes and open database connections. The problem with this model is that it requires your code to include vendor-specific class names, database URLs, and possibly other properties, such as machine names and port numbers.

With JDBC 2.0 data source functionality, using 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 allows your application code for opening database connections to be portable to other environments.

Data Source 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 an `OracleConnection` instance, optionally taking a user name and password as input.

To use other values, you can set properties using appropriate setter methods discussed in the next section. For alternative user names and passwords, you can also use the `getConnection()` signature that takes these 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.

Data Source Properties

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 14-1](#) and [Table 14-2](#) document `OracleDataSource` properties. The properties in [Table 14-1](#) are standard properties according to the Sun Microsystems specification. (Be aware, however, that Oracle does not implement the standard `roleName` property.) The properties in [Table 14-2](#) are Oracle extensions.

Table 14-1 Standard Data Source Properties

Name	Type	Description
<code>databaseName</code>	<code>String</code>	name of the particular database on the server; also known as the "SID" in Oracle terminology
<code>dataSourceName</code>	<code>String</code>	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)
<code>description</code>	<code>String</code>	description of the data source
<code>networkProtocol</code>	<code>String</code>	network protocol for communicating with the server; for Oracle, this applies only to the OCI drivers and defaults to <code>tcp</code> (Other possible settings include <code>ipc</code> . See the <i>Oracle Net Services Administrator's Guide</i> for more information.)
<code>password</code>	<code>String</code>	login password for the user name
<code>portNumber</code>	<code>int</code>	number of the port where the server listens for requests
<code>serverName</code>	<code>String</code>	name of the database server
<code>user</code>	<code>String</code>	name for the login account

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 that there is no `getPassword()` method, for security reasons.

Table 14–2 Oracle Extended Data Source Properties

Name	Type	Description
<code>driverType</code>	String	This designates the Oracle JDBC driver type as either <code>oci</code> , <code>thin</code> , or <code>kprb</code> (server-side internal).
<code>tnsEntry</code>	String	This is the TNS entry name, relevant only for the OCI driver. It assumes an Oracle client installation with a <code>TNS_ADMIN</code> environment variable that is set appropriately. Enable this <code>OracleXADataSource</code> property when using the HeteroRM XA feature with the OCI driver, to access Oracle pre-8.1.6 databases and higher. The HeteroRM XA feature is described in "OCI HeteroRM XA" on page 16-19. If the <code>tnsEntry</code> property is not set when using the HeteroRM XA feature, an <code>SQLException</code> with error code <code>ORA-17207</code> is thrown.

Table 14–2 Oracle Extended Data Source Properties (Cont.)

Name	Type	Description
url	String	This is 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 <code>tnsEntry</code> and <code>driverType</code> properties and the standard <code>portNumber</code> , <code>networkProtocol</code> , <code>serverName</code> , and <code>databaseName</code> properties.
nativeXA	boolean	Enable this <code>OracleXADataSource</code> property when using the HeteroRM XA feature with the OCI driver, to access Oracle pre-8.1.6 databases and higher. The HeteroRM XA feature is described in " OCI HeteroRM XA " on page 16-19. If the <code>nativeXA</code> property is enabled, be sure to set the <code>tnsEntry</code> property as well. This <code>DataSource</code> property defaults to <code>false</code> .

Note: Since `nativeXA` performs better than `JavaXA`, use `nativeXA` whenever possible.

The `OracleDataSource` class implements the following `setXXX()` and `getXXX()` methods for the Oracle extended properties:

- `public synchronized void setDriverType(String dt)`
- `public synchronized String getDriverType()`
- `public synchronized void setURL(String url)`
- `public synchronized String getURL()`
- `public synchronized void setTNSEntryName(String tns)`
- `public synchronized String getTNSEntryName()`
- `public synchronized void setNativeXA(boolean nativeXA)`
- `public synchronized boolean getNativeXA()`

If you are using the server-side internal driver—`driverType` property is set to `kprb`—then any other property settings are ignored.

If you are using a Thin or OCI driver, 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.

Creating a Data Source Instance and Connecting (without JNDI)

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("oci8");
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:

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

For a complete sample program, see ["Data Source without JNDI—DataSource.java"](#) on page 20-85.

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.

For a complete sample, see ["Data Source with JNDI—DataSourceJNDI.java"](#) on page 20-86.

Note: Creating and registering data sources is typically handled by a JNDI administrator, not in a JDBC application.

Initialize Connection Properties

Create an `OracleDataSource` instance, and then initialize its connection properties as appropriate, as in the following example:

```
...  
OracleDataSource ods = new OracleDataSource();  
  
ods.setDriverType("oci8");  
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 that 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 name space 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.

Register the Data Source for the Oracle9i JNDI

To bind the `OracleDataSource`, `OracleConnectionPoolDataSource` and `OracleXADataSource` instances into JNDI, use the `bindds` command of the Oracle9i `sess_sh` (session shell) tool. The `sess_sh` tool is furnished with Oracle9i as an interactive interface to the session namespace of a database instance. The `sess_sh` tool offers shell commands that give the session namespace much of the "look and feel" of a UNIX file system. (Once you start the `sess_sh` tool, a `$` command-line prompt appears.) See the *Oracle9i Java Tools Reference* manual for a complete description on using the `bindds` command.

Use the following syntax to bind a data source in an Oracle9i JNDI namespace:

```
$ bindds <datasource_name> [options]
```

Table 14-3 describes the `bindds` command options.

Table 14-3 Possible Data Source Options Using the `bindds` Command

Option	Description
<code>-h, -help</code>	prints out information about <code>bindds</code> options
<code>-version</code>	prints out version information
<code>-describe</code>	prints out a description of what the command does
<code>-rebind</code>	forces a bind if the data source object is already bound
<code>-g, -G, -grant</code> <code><schema1,schema2...></code>	adds read or execute schemas to the new object and directory
<code>-rg, -rG, -recursiveGrant</code> <code><schema1,schema2...></code>	adds read or execute schemas to the new data source object and directory
<code>-dstype <pool/xa/jta></code>	binds either <code>OracleDataSource</code> , <code>OraclePooled</code> , <code>OracleXA</code> , or <code>OracleJTADDataSource</code> (default is <code>OracleDataSource</code>)
<code>-url <jdbc_url></code>	specifies the URL, for example: <code>jdbc:oracle:oci:@</code>
<code>-host <host_name></code>	specifies the fully qualified host name of the data source
<code>-port <port_number></code>	specifies the port on which this data source listens
<code>-sid <database_name></code>	specifies the database ID
<code>-driver <JDBC_driver></code>	specifies a JDBC driver, such as <code>KPRB</code> , <code>OCI</code> , or <code>Thin</code>
<code>-dblink <dblink></code>	specifies the fully qualified database link
<code>-protocol <protocol></code>	specifies the protocol used to connect to the datasource
<code>-tns <tnsEntry></code>	specifies the Transparent Network Substrate (TNS) entry in the <code>tnsnames.ora</code> file (see the <i>Oracle Net Services Administrator's Guide</i>)

Table 14–3 Possible Data Source Options Using the `bindds` Command (Cont.)

Option	Description
<code>-u <user></code> , <code>-user <user></code>	specifies the user name for connection to the data source
<code>-p <pwd></code> , <code>-password <pwd></code>	specifies the password for the user name

The following `bindds` example creates an Oracle data source object that is bound as `jdbcdcb` (the logical name) into an Oracle9i JNDI namespace.

```
$ bindds jdbcdc -url jdbc:oracle:oci8:@ -u scott -p tiger
```

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 simply a Java Object) to a new `OracleDataSource` instance 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();
...
```

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 allows 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 log stream normally maintained 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.
-
-

Connection Pooling

Connection pooling in the JDBC 2.0 extension API is a framework for caching database connections. This allows reuse of physical connections and reduced overhead for your application. Connection pooling functionality minimizes expensive operations in the creation and closing of sessions.

The following are central concepts:

- *Connection pool data sources*—similar in concept and functionality to the data sources described previously, but with methods to return *pooled connection* instances, instead of normal connection instances.
- *Pooled connections*—a pooled connection instance represents a single physical connection to a database, remaining open during use by a series of *logical connection instances*.

A logical connection instance is a simple connection instance (such as a standard `Connection` instance or an `OracleConnection` instance) returned by a pooled connection instance. Each logical connection instance acts as a temporary handle to the physical connection represented by the pooled connection instance.

For connection pooling information specific to OCI drivers, see "[OCI Driver Connection Pooling](#)" on page 16-2. For further introductory and general information about connection pooling, refer to the Sun Microsystems specification for the JDBC 2.0 Optional Package.

Note: The concept of connection pooling is not relevant to the server-side internal driver, where you are simply using the default connection, and is only relevant to the server-side Thin driver within a single session.

Connection Pooling Concepts

If you do not use connection pooling, each connection instance (`java.sql.Connection` or `oracle.jdbc.OracleConnection` instance) encapsulates its own physical database connection. When you call the `close()` method of the connection instance, the physical connection itself is closed. This is true whether you obtain the connection instance through the JDBC 2.0 data source facility described under "[Data Sources](#)" on page 14-2, or through the `DriverManager` facility described under "[Open a Connection to a Database](#)" on page 3-3.

With connection pooling, an additional step allows physical database connections to be reused by multiple logical connection instances, which are temporary handles to the physical connection. Use a connection pool data source to return a pooled connection, which is what encapsulates the physical database connection. Then use the pooled connection to return JDBC connection instances (one at a time) that each act as a temporary handle.

Closing a connection instance that was obtained from a pooled connection does *not* close the physical database connection. It does, however, free the resources of the connection instance, clear the state, close statement objects created from the connection instance, and restore the defaults for the next connection instance that will be created.

To actually close the physical connection, you must invoke the `close()` method of the pooled connection. This would typically be performed in the middle tier.

Connection Pool Data Source Interface and Oracle Implementation

The `javax.sql.ConnectionPoolDataSource` interface outlines standard functionality of connection pool data sources, which are factories for pooled connections. The overloaded `getPooledConnection()` method returns a pooled connection instance and optionally takes a user name and password as input:

```
public interface ConnectionPoolDataSource
{
    PooledConnection getPooledConnection() throws SQLException;
    PooledConnection getPooledConnection(String user, String password)
        throws SQLException;
    ...
}
```

Oracle JDBC implements the `ConnectionPoolDataSource` interface with the `oracle.jdbc.pool.OracleConnectionPoolDataSource` class. This class also extends the `OracleDataSource` class, so it includes all the connection properties and getter and setter methods described in "[Data Source Properties](#)" on page 14-4.

The `OracleConnectionPoolDataSource` class `getPooledConnection()` methods return the Oracle implementation of pooled connection instances, which are `OraclePooledConnection` instances (as discussed in the next section).

Note: You can register connection pool data sources in JNDI using the same naming conventions as discussed for non-pooling data sources in ["Register the Data Source"](#) on page 14-9.

Pooled Connection Interface and Oracle Implementation

A pooled connection instance encapsulates a physical connection to a database. This database would be the one specified in the connection properties of the connection pool data source instance used to produce the pooled connection instance.

A pooled connection instance is an instance of a class that implements the standard `javax.sql.PooledConnection` interface. The `getConnection()` method specified by this interface returns a logical connection instance that acts as a temporary handle to the physical connection, as opposed to encapsulating the physical connection, as does a non-pooling connection instance:

```
public interface PooledConnection
{
    Connection getConnection() throws SQLException;
    void close() throws SQLException;
    void addConnectionEventListener(ConnectionEventListener listener) ... ;
    void removeConnectionEventListener(ConnectionEventListener listener);
    void setStmtCacheSize(int size);
    void setStmtCacheSize(int size, boolean clearMetaData);
    int getStmtCacheSize();
}
```

(Event listeners are used in connection caching and are discussed in ["Typical Steps in Using a Connection Cache"](#) on page 14-21.)

Oracle JDBC implements the `PooledConnection` interface with the `oracle.jdbc.pool.OraclePooledConnection` class. The `getConnection()` method returns an `OracleConnection` instance.

A pooled connection instance will typically be asked to produce a series of connection instances during its existence, but only one of these connection instances can be open at any particular time.

Each time a pooled connection instance `getConnection()` method is called, it returns a new connection instance that exhibits the default behavior, and it closes any previous connection instance that still exists and has been returned by the same pooled connection instance. You should explicitly close any previous connection instance before opening a new one, however.

Calling the `close()` method of a pooled connection instance closes the physical connection to the database. The middle-tier layer typically performs this.

The `OraclePooledConnection` class includes methods to enable statement caching for a pooled connection. The cache for statements is maintained for the pooled connection as a whole, and all logical connections obtained from the pooled connection share it. Therefore, when statement caching is enabled, a statement you create on one logical connection can be re-used on another logical connection. For the same reason, you cannot enable or disable statement caching on individual logical connections. This function applies to both implicit and explicit statement caching.

The following are `OraclePooledConnection` method definitions for statement caching:

```
public void setStmtCacheSize (int size)
    throws SQLException

public void setStmtCacheSize (int size, boolean clearMetaData)
    throws SQLException

public int getStmtCacheSize()
```

See [Chapter 13, "Statement Caching"](#), for more details on statement caching.

Creating a Connection Pool Data Source and Connecting

This section contains an example of the most basic use of a connection pool data source to connect to a database without using JNDI functionality. You could optionally use JNDI, binding the connection pool data source instance to a JNDI logical name, in the same way that you would for a generic data source instance (as "[Register the Data Source](#)" on page 14-9 illustrates).

Summary of Imports for Oracle Connection Pooling

You must import the following for Oracle connection pooling functionality:

```
import oracle.jdbc.pool.*;
```

This package contains the `OracleDataSource`, `OracleConnectionPoolDataSource`, and `OraclePooledConnection` classes, in addition to classes for connection caching and event-handling, which "[Connection Caching](#)" on page 14-18 discusses.

Oracle Connection Pooling Code Sample

This example first creates an `OracleConnectionPoolDataSource` instance, next initializes its connection properties, then gets a pooled connection instance from the connection pool data source instance, and finally gets a connection instance from the pooled connection instance. (The `getPooledConnection()` method actually returns an `OraclePooledConnection` instance, but in this case only generic `PooledConnection` functionality is required.)

```
...
OracleConnectionPoolDataSource ocpds = new OracleConnectionPoolDataSource();

ocpds.setDriverType("oci8");
ocpds.setServerName("dlsun999");
ocpds.setNetworkProtocol("tcp");
ocpds.setDatabaseName("816");
ocpds.setPortNumber(1521);
ocpds.setUser("scott");
ocpds.setPassword("tiger");

PooledConnection pc = ocpds.getPooledConnection();

Connection conn = pc.getConnection();
...
```

For a complete sample program, see ["Pooled Connection—PooledConnection.java"](#) on page 20-89.

Connection Caching

Connection caching, generally implemented in a middle tier, is a means of keeping and using caches of physical database connections.

Connection caching uses the connection pooling framework—such as connection pool data sources and pooled connections—in much of its operations. "[Connection Pooling](#)", starting on page 14-13, describes this framework.

The JDBC 2.0 specification does not mandate a connection caching implementation, but Oracle provides a simple implementation to serve at least as an example.

This section is divided into the following topics:

- [Overview of Connection Caching](#)
- [Typical Steps in Using a Connection Cache](#)
- [Oracle Connection Cache Specification: OracleConnectionCache Interface](#)
- [Oracle Connection Cache Implementation: OracleConnectionCacheImpl Class](#)
- [Oracle Connection Event Listener: OracleConnectionEventListener Class](#)

Note: The concept of connection caching is not relevant to the server-side internal driver, where you are simply using the default connection, and is only relevant to the server-side Thin driver within a single session.

Overview of Connection Caching

Each connection cache is represented by an instance of a *connection cache class* and has an associated group of pooled connection instances. For a single connection cache instance, the associated pooled connection instances must all represent physical connections to the same database and schema. Pooled connection instances are created as needed, which is whenever a connection is requested and the connection cache does not have any free pooled connection instances. A "free" pooled connection instance is one that currently has no logical connection instance associated with it; in other words, a pooled connection instance whose physical connection is not being used.

Basics of Setting Up a Connection Cache

The middle tier, in setting up a connection cache, will create an instance of a connection cache class and set its data source connection properties as

appropriate—for example, `serverName`, `databaseName`, or URL. Recall that a connection cache class extends a data source class. For information about data source properties, see ["Data Source Properties"](#) on page 14-4.

An example of a connection cache class is `OracleConnectionCacheImpl`. How to instantiate this class and set its connection properties is described in ["Instantiating OracleConnectionCacheImpl and Setting Properties"](#) on page 14-25. This class extends the `OracleDataSource` class and so includes the setter methods to set connection properties to specify the database to connect to. All the pooled connection instances in the cache would represent physical connections to this same database, and in fact to the same schema.

Once the middle tier has created a connection cache instance, it can optionally bind this instance to JNDI as with any data source instance, which is described in ["Register the Data Source"](#) on page 14-9.

Basics of Accessing the Connection Cache

A JDBC application must retrieve a connection cache instance to use the cache. This is typically accomplished through the middle tier, often using a JNDI lookup. In a connection caching scenario, a JNDI lookup would return a connection cache instance instead of a generic data source instance. Because a connection cache class extends a data source class, connection cache instances include data source functionality.

Executing a JNDI lookup is described in ["Open a Connection"](#) on page 14-11.

If JNDI is not used, the middle tier will typically have some vendor-specific API through which a connection cache instance is retrieved for the application.

Basics of Opening Connections

A connection cache class, as with a pooled connection class, has a `getConnection()` method. The `getConnection()` method of a connection cache instance returns a logical connection to the database and schema associated with the cache. This association is through the connection properties of the connection cache instance, as typically set by the middle tier.

Whenever a JDBC application wants a connection to a database in a connection caching scenario, it will call the `getConnection()` method of the connection cache instance associated with the database.

This `getConnection()` method checks if there are any free pooled connection instances in the cache. If not, one is created. Then a logical connection instance will

be retrieved from a previously existing or newly created pooled connection instance, and this logical connection instance will be supplied to the application.

Basics of Closing Connections: Use of Connection Events

JDBC uses JavaBeans-style events to keep track of when a physical connection (pooled connection instance) can be returned to the cache or when it should be closed due to a fatal error. When a JDBC application calls the `close()` method of a logical connection instance, an event is triggered and communicated to the event listener or listeners associated with the pooled connection instance that produced the logical connection instance. This triggers a connection-closed event and informs the pooled connection instance that its physical connection can be reused. Essentially, this puts the pooled connection instance and its physical connection back into the cache.

The point at which a connection event listener is created and registered with a pooled connection instance is implementation-specific. This could happen, for example, when the pooled connection instance is first created or each time the logical connection associated with it is closed.

It is also possible for the cache class to implement the connection event listener class. In this case, the connection event listener is part of the connection cache instance. (This is not the case in the Oracle sample implementation.) Even in this case, however, an explicit association must be made between the connection event listener and each pooled connection instance.

Implementation Scenarios

Middle-tier developers have the option of implementing their own connection cache class and connection event listener class.

For convenience, however, Oracle provides the following, all in the `oracle.jdbc.pool` package:

- a connection cache interface: `OracleConnectionCache`
- a connection cache class: `OracleConnectionCacheImpl`
- a connection event listener class: `OracleConnectionEventListener`

The `OracleConnectionCacheImpl` class is a simple connection cache class implementation that Oracle supplies as an example, providing sufficient but minimal functionality. It implements the `OracleConnectionCache` interface and uses instances of the `OracleConnectionEventListener` class for connection events.

If you want more functionality than `OracleConnectionCacheImpl` has to offer but still want to use `OracleConnectionEventListener` for connection events, then you can create your own class that implements `OracleConnectionCache`.

Or you can create your own connection cache class and connection event listener class from scratch.

Typical Steps in Using a Connection Cache

This section lists the general steps in how a JDBC application and middle-tier will use a connection cache in opening and closing a logical connection.

Preliminary Steps in Connection Caching

Presume the following has already been accomplished:

1. The middle tier has created a connection cache instance, as described in ["Basics of Setting Up a Connection Cache"](#) on page 14-18.
2. The middle tier has provided connection information to the connection cache instance for the database and schema that will be used. This can be accomplished when constructing the connection cache instance.
3. The application has retrieved the connection cache instance, as described in ["Basics of Accessing the Connection Cache"](#) on page 14-19.

General Steps in Opening a Connection

Once the JDBC application has access to the connection cache instance, the application and middle tier perform the following steps to produce a logical connection instance for use by the application:

1. The application requests a connection through the `getConnection()` method of the connection cache instance. No input is necessary, because a connection cache instance is already associated with a particular database and schema.
2. The connection cache instance examines its cache as follows: a) to see if there are any pooled connection instances in the cache yet; and b) if so, if any are free—that is, to see if there is at least one pooled connection instance that currently has no logical connection instance associated with it.
3. The connection cache instance chooses an available pooled connection instance or, if none is available, might create a new one (this is implementation-specific). In creating a pooled connection instance, the connection cache instance can specify connection properties according to its own connection properties,

because the pooled connection instance will be associated with the same database and schema.

Note: Exactly what happens in a situation where no pooled connection instances are available depends on the implementation schemes and whether the cache is limited to a maximum number of pooled connections. For the Oracle sample implementation, this is discussed in "[Schemes for Creating New Pooled Connections in the Oracle Implementation](#)" on page 14-27.

4. Depending on the situation and implementation, the connection cache instance creates a connection event listener (a process that associates the listener with the connection cache instance) and associates the listener with the chosen or newly created pooled connection instance. The association with the pooled connection instance is accomplished by calling the standard `addConnectionEventListener()` method specified by the `PooledConnection` interface. This method takes the connection event listener instance as input. If the connection cache class implements the connection event listener class, then the argument to the `addConnectionEventListener()` method would be the `this` object.

In some implementations, the creation and association of the connection event listener can occur only when the pooled connection instance is first created. In the Oracle sample implementation, this also occurs each time a pooled connection instance is reused.

Note that in being associated with both the connection cache instance and a pooled connection instance, the connection event listener becomes the bridge between the two.

5. The connection cache instance gets a logical connection instance from the chosen or newly created pooled connection instance, using the pooled connection `getConnection()` method.

No input is necessary to `getConnection()`, because a pooled connection instance is already associated with a particular database and schema.

6. The connection cache instance passes the logical connection instance to the application.

The JDBC application uses this logical connection instance as it would any other connection instance.

General Steps in Closing a Connection

Once the JDBC application has finished using the logical connection instance, its associated pooled connection instance can be returned to the connection cache (or closed, as appropriate, if a fatal error occurred). The application and middle tier perform the following steps to accomplish this:

1. The application calls the `close()` method on the logical connection instance (as it would with any connection instance).
2. The pooled connection instance that produced the logical connection instance triggers an event to the connection event listener or listeners associated with it (associated with it through previous calls by the connection cache instance to the pooled connection instance `addConnectionEventListener()` method).
3. The connection event listener performs one of the following:
 - It puts the pooled connection instance back into the cache and flags it as available (typical).

or:

- It closes the pooled connection instance (if a fatal error occurred during use of its physical connection).

The connection event listener will typically perform these steps by calling methods of the connection cache instance, which is implementation-specific. For the Oracle sample implementation, these functions are performed by methods specified in the `OracleConnectionCache` interface, as discussed in "[Oracle Connection Cache Specification: OracleConnectionCache Interface](#)" on page 14-24.

4. Depending on the situation and implementation, the connection cache instance disassociates the connection event listener from the pooled connection instance. This is accomplished by calling the standard `removeConnectionEventListener()` method specified by the `PooledConnection` interface.

In some implementations, this step can be performed only when a pooled connection instance is closed, either because of a fatal error or because the application is finished with the physical connection. In the Oracle sample implementation, however, the connection event listener is disassociated with the pooled connection instance each time the pooled connection is returned to the available cache (because in the Oracle implementation, a connection event listener is associated with the pooled connection instance whenever it is reused).

Oracle Connection Cache Specification: OracleConnectionCache Interface

Middle-tier developers are free to implement their own connection caching scheme as desired, but Oracle offers the `OracleConnectionCache` interface, which you can implement in a connection cache class and which uses instances of the `OracleConnectionEventListener` class for its listener functionality.

In addition, Oracle offers a class that implements this interface, `OracleConnectionCacheImpl`, which can be used as is. This class also extends the `OracleDataSource` class and, therefore, includes a `getConnection()` method. For more information about this class, see "[Oracle Connection Cache Implementation: OracleConnectionCacheImpl Class](#)" on page 14-25.

These Oracle classes and interfaces are all in the `oracle.jdbc.pool` package.

The `OracleConnectionCache` interface specifies the following methods (in addition to data source methods that it inherits), to be implemented in a connection cache class:

- `reusePooledConnection()`: Takes a pooled connection instance as input and returns it to the cache of available pooled connections (essentially, the available physical connections).

This method would be invoked by a connection event listener after a JDBC application has finished using the logical connection instance provided by the pooled connection instance (through previous use of the pooled connection `getConnection()` method).

- `closePooledConnection()`: Takes a pooled connection instance as input and closes it.

A connection event listener would invoke this method after a fatal error has occurred through the logical connection instance provided by the pooled connection instance. The listener would call `closePooledConnection()`, for example, if it notices a server crash.

- `close()`: Closes the connection cache instance, after the application has finished using connection caching with the associated database.

The functionality of the `reusePooledConnection()` and `closePooledConnection()` methods is an implementation of some of the steps described generally in "[General Steps in Closing a Connection](#)" on page 14-23.

Oracle Connection Cache Implementation: OracleConnectionCacheImpl Class

Oracle offers a sample implementation of connection caching and connection event listeners, providing the `OracleConnectionCacheImpl` class. This class implements the `OracleConnectionCache` interface (which you can optionally implement yourself in some other connection cache class) and uses instances of the `OracleConnectionEventListener` class for listener functionality.

These Oracle classes and interfaces are all in the `oracle.jdbc.pool` package.

If you use the `OracleConnectionCacheImpl` class for your connection caching functionality, you should be familiar with the following topics, discussed immediately below:

- [Instantiating OracleConnectionCacheImpl and Setting Properties](#)
- [Setting a Maximum Number of Pooled Connections](#)
- [Setting a Minimum Number of Pooled Connections](#)
- [Schemes for Creating New Pooled Connections in the Oracle Implementation](#)
- [Additional OracleConnectionCacheImpl Methods](#)

Instantiating OracleConnectionCacheImpl and Setting Properties

A middle tier that uses the Oracle implementation of connection caching can construct an `OracleConnectionCacheImpl` instance and set its connection properties in one of three ways:

- It can use the `OracleConnectionCacheImpl` constructor that takes an existing connection pool data source as input. This is convenient if the middle tier has already created a connection pool data source instance and set its connection properties. For example, where `cpds` is a connection pool data source instance:

```
OracleConnectionCacheImpl ocacheimpl = new OracleConnectionCacheImpl(cpds);
```

or:

- It can use the default `OracleConnectionCacheImpl` constructor (which takes no input) and then the `setConnectionPoolDataSource()` method, which takes an existing connection pool data source instance as input. Again, this is convenient if the middle tier already has a connection pool data source instance with its connection properties set. For example, where `cpds` is a connection pool data source instance:

```
OracleConnectionCacheImpl ocacheimpl = new OracleConnectionCacheImpl();
```

```
ocacheimpl.setConnectionPoolDataSource(cpds);
```

Notes:

- You can also use the `setConnectionPoolDataSource()` method to override a previously set pooled connection data source or previously set connection properties.
 - If you call `setConnectionPoolDataSource()` when there is already a connection pool data source with associated logical connections in use, then an exception will be thrown if the new connection pool data source specifies a different database schema than the old connection pool data source.
-

or:

- It can use the default `OracleConnectionCacheImpl` constructor and then set the properties individually, using setter methods. For example:

```
OracleConnectionCacheImpl ocacheimpl = new OracleConnectionCacheImpl();  
  
ocacheimpl.setDriverType("oci8");  
ocacheimpl.setServerName("dlsun999");  
ocacheimpl.setNetworkProtocol("tcp");  
ocacheimpl.setDatabaseName("816");  
ocacheimpl.setPortNumber(1521);  
ocacheimpl.setUser("scott");  
ocacheimpl.setPassword("tiger");
```

This is equivalent to setting properties in any generic data source or connection pool data source, as discussed in ["Initialize Connection Properties"](#) on page 14-8.

Setting a Maximum Number of Pooled Connections

In any connection caching implementation, the middle-tier developer must decide whether there should be a maximum number of pooled connections in the cache, and how to handle situations where no pooled connections are available and the maximum number has been reached.

The `OracleConnectionCacheImpl` class includes a maximum cache size that you can set using the `setMaxLimit()` method (taking an `int` as input). The default value is 1.

The following is an example that presumes `ocacheimpl` is an `OracleConnectionCacheImpl` instance:

```
ocacheimpl.setMaxLimit(10);
```

This example limits the cache to a maximum size of ten pooled-connection instances.

Setting a Minimum Number of Pooled Connections

Just as the middle-tier developer can set the maximum number of pooled connections, you can also determine if there should be a minimum number of pre-spawned pooled connections in the cache. The minimum number is passed as an argument to the `setMinLimit()` method. If the cache doesn't have the specified number of pooled connections instances, the cache will create the new spooled-connection instances, not exceeding the specified minimum limit. The cache always keeps the minimum number of pooled connections open whether the connections are active or idle.

The following is an example that presumes `ocacheimpl` is an `OracleConnectionCacheImpl` instance:

```
ocacheimpl.setMinLimit(3);
```

The cache, in this example, always has a minimum of three pooled-connection instances.

Schemes for Creating New Pooled Connections in the Oracle Implementation

The `OracleConnectionCacheImpl` class supports three *connection cache schemes*. Use these schemes in situations where (1) the application has requested a connection, (2) all existing pooled connections are in use, and (3) the maximum number of pooled connections in the cache have been reached.

- dynamic

In this default scheme, you can create new pooled connections above and beyond the maximum limit, but each one is automatically closed and freed as soon as the logical connection instance that it provided is no longer in use. (As opposed to the normal scenario when a pooled connection instance is finished being used, where it is returned to the available cache.)

- **fixed with no wait**

In this scheme, the maximum limit cannot be exceeded. Requests for connections when the maximum has already been reached will return `null`.

- **fixed wait**

Same as the "fixed with no wait" scheme except that a request for a new connection will wait if the limit for the number of connections has been reached. In this case, the connection request waits until another client releases a connection.

Set the cache scheme by invoking the `setCacheScheme()` method of the `OracleConnectionCacheImpl` instance. Use one of the following class static constants as input:

- `DYNAMIC_SCHEME`
- `FIXED_RETURN_NULL_SCHEME`
- `FIXED_WAIT_SCHEME`

For example, presuming `ocacheimpl` is an `OracleConnectionCacheImpl` instance:

```
ocacheimpl.setCacheScheme(OracleConnectionCacheImpl.FIXED_RETURN_NULL_SCHEME);
```

An example of each scheme is available in the Sample Applications chapter—see ["Oracle Connection Cache \(dynamic\)—CCache1.java"](#) on page 20-101 and ["Oracle Connection Cache \("fixed with no wait"\)—CCache2.java"](#) on page 20-103.

Additional OracleConnectionCacheImpl Methods

In addition to the key methods already discussed in ["Oracle Connection Cache Specification: OracleConnectionCache Interface"](#) on page 14-24, the following `OracleConnectionCacheImpl` methods may be useful:

- `getActiveSize()`: Returns the number of currently active pooled connections in the cache (pooled connection instances with an associated logical connection instance being used by the JDBC application).
- `getCacheSize()`: Returns the total number of pooled connections in the cache, both active and inactive.

Oracle Connection Event Listener: OracleConnectionEventListener Class

This section discusses `OracleConnectionEventListener` functionality by summarizing its constructors and methods.

Instantiating an Oracle Connection Event Listener

In the Oracle implementation of connection caching, an `OracleConnectionCacheImpl` instance constructs an Oracle connection event listener, specifying the connection cache instance itself (its `this` instance) as the constructor argument. This instance associates the connection event listener with the connection cache instance.

In general, however, the `OracleConnectionEventListener` constructor can take any data source instance as input. For example, where `ds` is a generic data source:

```
OracleConnectionEventListener ocel = new OracleConnectionEventListener(ds);
```

There is also a default constructor that takes no input and can be used in conjunction with the `OracleConnectionEventListener` class `setDataSource()` method:

```
OracleConnectionEventListener ocel = new OracleConnectionEventListener();  
...  
ocel.setDataSource(ds);
```

The input can be any kind of data source, including an `OracleConnectionCacheImpl` instance (because that class extends `OracleDataSource`).

Oracle Connection Event Listener Methods

This section summarizes the methods of the `OracleConnectionEventListener` class:

- `setDataSource()` (previously discussed): Used to input a data source to the connection event listener, in case one was not provided when constructing the listener. This can take any type of data source as input.
- `connectionClosed()`: Invoked when the JDBC application calls `close()` on its representation of the connection.
- `connectionErrorOccurred()`: Invoked when a fatal connection error occurs, just before a `SQLException` is issued to the application.

Distributed Transactions

This chapter discusses the Oracle JDBC implementation of distributed transactions. These are multi-phased transactions, often using multiple databases, that must be committed in a coordinated way. There is also related discussion of XA, which is a general standard (not specific to Java) for distributed transactions.

The following topics are discussed:

- [Overview](#)
- [XA Components](#)
- [Error Handling and Optimizations](#)
- [Implementing a Distributed Transaction](#)

Note: This chapter discusses features of the JDBC 2.0 Optional Package, formerly known as the JDBC 2.0 Standard Extension API, which is available through the `javax` packages from Sun Microsystems. The Optional Package is not part of the standard JDK, but relevant packages are included with the Oracle JDBC `classes111.zip` and `classes12.zip` files.

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).

For information on the OCI-specific HeteroRM XA feature, see "[OCI HeteroRM XA](#)" on page 16-19.

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, described under "[Connection Pooling](#)" on page 14-13. 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.)

The remainder of this overview covers the following topics:

- [Distributed Transaction Components and Scenarios](#)
- [Distributed Transaction Concepts](#)
- [Oracle XA Packages](#)

For further introductory and general information about distributed transactions and XA, refer to the Sun Microsystems specifications for the JDBC 2.0 Optional Package and the Java Transaction API.

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 Java Transaction API functionality—to coordinate the individual transactions.

Many vendors will offer XA-compliant JTA modules. This includes Oracle, which is developing a JTA module based on the Oracle implementation of XA discussed below.

- 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 file `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

Software that uses distributed transactions cannot use normal connection instance `COMMIT`, `auto-commit`, or `ROLLBACK` functionality, because all `COMMIT` or `ROLLBACK` operations in a distributed transaction must be coordinated. Any attempt to use the `commit()` or `rollback()` method or enable the `auto-commit` flag of a connection instance would result in a SQL exception.

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 (database) that will be used in the distributed transaction. You will typically create XA data source instances (using the class constructor) 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 (one at a time), as with pooled connection instances.

You will typically get an XA connection instance from an XA data source instance (using a `get` method) 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 (multiple physical connections) in the same database.

XA connections produce XA resource 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 XA resource instance from each XA connection instance (using a `get` method), typically in your middle-tier software. There is a one-to-one correlation between XA resource instances and XA connection instances; equivalently, there is a one-to-one correlation between XA resource instances and Oracle sessions (physical connections).

In a typical scenario, the middle-tier component will hand off XA resource instances to the transaction manager, for use in coordinating distributed transactions.

Because each XA resource instance corresponds to a single Oracle session, there can be only a single active transaction branch associated with an XA resource instance at any given time. There can be additional suspended transaction branches, however—see ["XA Resource Method Functionality and Input Parameters"](#) on page 15-9.

Each XA resource 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 XA resource 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 XA resource instance. Once the transaction manager sees that the operations of each transaction branch have prepared successfully (essentially, that the databases can be accessed without error), it will issue a `COMMIT` to each XA resource 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 XA resource instances associated with a given distributed transaction would have a transaction ID that includes the same distributed transaction ID component.

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` (`OracleXid` and `OracleXAException` classes)
- `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—standard XA interfaces specified in the JDBC 2.0 Optional Package, and the Oracle classes that implement them. The following topics are covered:

- [XA Data Source Interface and Oracle Implementation](#)
- [XA Connection Interface and Oracle Implementation](#)
- [XA Resource Interface and Oracle Implementation](#)
- [XA Resource Method Functionality and Input Parameters](#)
- [XA ID Interface and Oracle Implementation](#)

XA Data Source 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), so include all the connection properties described in "[Data Source Properties](#)" on page 14-4.

The `OracleXADataSource` class `getXAConnection()` methods return the Oracle implementation of XA connection instances, which are `OracleXAConnection` instances (as the next section discusses).

Note: You can register XA data sources in JNDI using the same naming conventions as discussed previously for non-pooling data sources in ["Register the Data Source"](#) on page 14-9.

XA Connection 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 XA resource 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 listed in ["Pooled Connection Interface and Oracle Implementation"](#) on page 14-15.

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 XA resource instance, which is an `OracleXAResource` instance (as the next section discusses). 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.

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. It is advisable to explicitly close any previous connection instance before opening a new one, however.

Calling the `close()` method of an XA connection instance closes the physical connection to the database. This is typically performed in the middle tier.

XA Resource Interface and Oracle Implementation

The transaction manager uses XA resource instances to coordinate all the transaction branches that constitute a distributed transaction.

Each XA resource 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 XA resource instance. (Essentially, 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.

["XA Resource Method Functionality and Input Parameters"](#) on page 15-9 further discusses this.

Notes:

- Because there must always be a one-to-one correlation between XA connection instances and XA resource instances, an XA resource instance is implicitly closed when the associated XA connection instance is closed.
 - If a transaction is opened by a given XA resource instance, it must also be closed by the same XA resource instance.
-
-

An XA resource instance is an instance of a class that implements the standard `javax.transaction.xa.XAResource` interface:

```
public interface XAResource
{
    void commit(Xid xid, boolean onePhase) throws XAException;
    void end(Xid xid, int flags) throws XAException;
    void forget(Xid xid) throws XAException;
    int prepare(Xid xid) throws XAException;
    Xid[] recover(int flag) throws XAException;
    void rollback(Xid xid) throws XAException;
    void start(Xid xid, int flags) throws XAException;
    boolean isSameRM(XAResource xares) throws XAException;
}
```

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 `OracleXAConnection` class `getXAResource()` method is called, and it is the Oracle JDBC driver that associates an XA resource instance with a connection instance and the transaction branch being executed through that connection.

This method is how an `OracleXAResource` instance is associated with a particular connection and with the transaction branch being executed in that connection.

XA Resource 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.

"[XA ID Interface and Oracle Implementation](#)" on page 15-13 discusses the `OracleXid` class and the standard interface upon which it is based.

Following is a description of key XA resource functionality, the methods used, and additional input parameters. Each of these methods throws an XA exception if an error is encountered. See "[XA Exception Classes and Methods](#)" on page 15-15.

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 can have one of the following values:

- `XAResource.TMNOFLAGS` (no special flag)—This is to 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.TMJJOIN`—This is to join subsequent operations in the session associated with this XA resource instance to the existing transaction branch specified by `xid`.
- `XAResource.TMRESUME`—This is to resume the transaction branch specified by `xid`. (It must first have been suspended.)

`TMNOFLAGS`, `TMJOIN`, and `TMRESUME` are defined as static members of the `XAResource` interface and `OracleXAResource` class.

Note: Instead of using the `start()` method with `TMRESUME`, the transaction manager can cast to an `OracleXAResource` instance and use the `resume(Xid xid)` method, an Oracle extension.

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 and are beyond the scope of this document. Refer to the Sun Microsystems specifications for the JDBC 2.0 Optional Package and the Java Transaction API.

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 an `OracleXAResource` instance 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.
-
-

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 executes only read-only operations such as `SELECT` statements.
- `XAResource.XA_OK`—This is returned if the transaction branch executes updates that are all prepared without error.

- `n/a` (no value returned)—No value is returned if the transaction branch executes 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 XA resource `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 (typical).

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: The `flag` parameter is ignored and therefore not implemented since the `scan` option (`flag` parameter) is not meaningful without a `count` parameter. See the Sun Microsystems *Java Transaction API (JTA)* Specification for more detail.

The resource manager returns zero or more `Xids` for the transaction branches that are currently in a prepared or heuristically completed state. If an error occurs during the operation, the resource manager throws the appropriate `XAException`.

Check for same RM To determine if two XA resource instances correspond to the same resource manager (database), call the `isSameRM()` method from one XA resource instance, specifying the other XA resource instance as input. In the following example, presume `xares1` and `xares2` are `OracleXAResource` instances:

```
boolean sameRM = xares1.isSameRM(xares2);
```

A transaction manager can use this method regarding certain Oracle optimizations, as "[Oracle XA Optimizations](#)" on page 15-17 explains.

XA ID 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 (4 bytes)
A format identifier specifies a Java transaction manager—for example, there could be a format identifier `ORCL`. This field *cannot* be null.
- global transaction identifier (64 bytes) (or "distributed transaction ID component", as discussed earlier)
- branch qualifier (64 bytes) (or "transaction branch ID component", as discussed earlier)

The 64-byte global transaction identifier value will be identical in the transaction IDs of all transaction branches belonging to the same distributed transaction. The overall transaction ID, however, 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: `OracleXid` is not required for Oracle XA resource calls. Instead, use any class that implements `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 has two focuses: 1) the functionality of XA exceptions and error handling; and 2) Oracle optimizations in its XA implementation. The following topics are covered:

- [XA Exception 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.

XA Exception Classes and Methods

XA methods throw XA exceptions, as opposed to general exceptions or SQL exceptions. 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 as follows by the Oracle JDBC driver:

```
public OracleXAException()
```

or:

```
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; refer to its Javadoc at the Sun Microsystems Web site for more information.

Mapping between Oracle Errors and XA Errors

Oracle errors correspond to XA errors in `OracleXAException` instances as documented in [Table 15-1](#).

Table 15-1 Oracle-XA Error Mapping

Oracle Error Code	XA Error Code
ORA 3113	<code>XAException.XAER_RMFAIL</code>
ORA 3114	<code>XAException.XAER_RMFAIL</code>
ORA 24756	<code>XAException.XAER_NOTA</code>
ORA 24764	<code>XAException.XA_HEURCOM</code>
ORA 24765	<code>XAException.XA_HEURRB</code>
ORA 24766	<code>XAException.XA_HEURMIX</code>
ORA 24767	<code>XAException.XA_RDONLY</code>
ORA 25351	<code>XAException.XA_RETRY</code>
all other ORA errors	<code>XAException.XA_RMERR</code>

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 XA resource instances associated with these branches are associated with the same resource manager.

In such a circumstance, the `prepare()` method of only one of these XA resource instances will return `XA_OK` (or failure); 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, if failure) the joined transaction through the XA resource instance that returned `XA_OK` (or failure).

The transaction manager can use the `OracleXAResource` class `isSameRM()` method to determine if two XA resource 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.

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.

In addition, if the code will run inside an Oracle database and access that database for SQL operations, you must import the following:

```
import oracle.jdbc.xa.server.*;
```

(And if you intend to access *only* the database in which the code runs, you would not need the `oracle.jdbc.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 XA resource method invocations and the creation of transaction IDs).

For brevity, the specifics of creating transaction IDs (in the `createID()` method) and performing SQL operations (in the `doSomeWork1()` and `doSomeWork2()` methods) are not shown here. The complete sample is in ["XA with Two-Phase Commit Operation—XA4.java"](#) on page 20-109.

For another complete sample, showing how to use XA resource functionality to suspend and resume a transaction, see "[XA with Suspend and Resume—XA2.java](#)" on page 20-105.

This example executes 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:oci8:@";
            String URL2 = "jdbc:oracle:thin:@(description=(address=(host=dlsun991)
                (protocol=tcp)(port=5521))(connect_data=(sid=rdbms2)))";

            DriverManager.registerDriver(new OracleDriver());
```

```
// You can put a database name after the @ sign in the connection URL.
Connection conna =
    DriverManager.getConnection (URL1, "scott", "tiger");

// Prepare a statement to create the table
Statement stmta = conna.createStatement ();

Connection connb =
    DriverManager.getConnection (URL2, "scott", "tiger");

// 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 (coll 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 (coll char(30))");
}
catch (SQLException e)
{
    // Ignore an error here too
}

// Create XADataSource instances and set properties.
OracleXADataSource oxds1 = new OracleXADataSource();
oxds1.setURL("jdbc:oracle:oci8:@");
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 coll from my_table");
while (rset.next())
    System.out.println("Coll is " + rset.getInt(1));

rset.close();
rset = null;

rset = stmtb.executeQuery ("select coll from my_tab");
while (rset.next())
    System.out.println("Coll 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...}  
  
}
```

JDBC OCI Extensions

This chapter describes the following OCI driver-specific features:

- [OCI Driver Connection Pooling](#)
- [Middle-Tier Authentication Through Proxy Connections](#)
- [OCI Driver Transparent Application Failover](#)
- [OCI HeteroRM XA](#)
- [Accessing PL/SQL Index-by Tables](#)

OCI Driver Connection Pooling

OCI driver connection pooling functionality, provided by the `OracleOCIConnectionPool` class, is part of the JDBC client. Enhanced connection pooling provides the following benefits:

- **Improved scalability** - The pooling granularity is superior to that provided by the `OraclePooledConnection` class since fewer physical connections are needed to support a large number of non-current, logical connections. This is valuable since physical connections are expensive. The physical connection of the `OraclePooledConnection` object is available for reuse after the application is done using it. Also, since the user session is not closed on the server-side once the `OraclePooledConnection` object is returned to the pool of available connection objects, every new call to the `getConnection()` method of the `OracleConnectionCacheImpl` class requires that the user remain the same. For a dedicated server instance, this results in the number of backend Oracle processes being reduced as the number of in-coming connections are also reduced. To boost performance, a physical connection is locked only for the duration of a call.
- **Uniform interface** - A single, uniform interface of connection pooling reduces overall code maintenance.
- **Flexible schemas** - Each `OracleOCIConnection` object can have a different user ID and therefore point to different schemas.
- **Dynamic configuration** - Ability to dynamically configure the connection pool.

Note: The existing connection support of mapping one JDBC user session to one physical connection, and the reuse of physical connection objects using the `OraclePooledConnection` class, is still supported. (See "[Connection Pooling](#)" on page 14-13 for details.) However, it is recommended that you use the improved functionality of the `OracleOCIConnectionPool` class instead.

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 OCI in Oracle9i allows applications to have many logical connections, all using a small set of physical connections. Each call on this logical connection will be routed on the physical connection that is available at that time. Call-duration based pooling of connections is a more scalable connection pooling solution.

For information about Oracle JDBC connection pooling and caching features that apply to all Oracle JDBC drivers, see [Chapter 14, "Connection Pooling and Caching"](#). See "[OCI Connection Pool—OCIConnectionPool.java](#)" on page 20-90.

OCI Driver Connection Pooling: Background

With the Oracle9i JDBC OCI driver, there are 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 backend 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` connection objects obtained from the `OracleOCIConnectionPool`. The connection pool itself is normally configured with a much smaller shared pool of physical connections, translating to a backend 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 backend Oracle processes.

OCI Driver Connection Pooling and MTS Compared

In some ways, what OCI driver connection pooling offers on the middle tier is similar to what MTS offers on the backend. OCI driver connection pooling makes a dedicated server instance behave as an MTS 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 MTS is that in case of MTS using 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 backend server pool.

Note that OCI connection pool is mainly beneficial only if the middle tier is multi-threaded. Each thread could maintain a session to the database. The actual connections to the database are maintained by the `OracleOCIConnectionPool`

and these connections (including the pool of dedicated database server processes) are shared among all the threads in the middle tier.

Stateless Sessions Compared to Stateful Sessions

OCI connection pooling offers stateless physical connections and stateful sessions. If you need to work with a stateless session behavior, you can use the `OracleConnectionCacheImpl` interface.

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` class 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` class 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

/*
 * For getting a connection to the database.
 *
 * Notes: This will take user and password use for the
 * OracleOCIConnectionPool() call unless setUser and setPassword
 * calls were made.

 * @return    connection object
 */

```

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.*;
```

The `oracle.jdbc.pool.*` package contains the `OracleDataSource`, `OracleConnectionPoolDataSource`, and `OracleOCIConnectionPool` classes, in addition to classes for connection caching and event-handling. The `oracle.jdbc.oci.*` package contains the `OracleOCIConnection` class and the `OracleOCIFailover` interface.

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=(sid=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 above 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=(sid=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` : When enabled, this attribute specifies 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 (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 the above 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" (case insensitive).

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 estimate and for statistical analysis.
- `int getActiveSize()`: Retrieves the number of physical connections that are open and busy. This should be used only as 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. See "Data Sources" on page 14-2 for database connection descriptions that apply to all JDBC drivers.

Since 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(String user, String password)`: Get a logical connection identified with the specified user and password, which can be different from that used for pool creation.
- `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.

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)
```

The following code shows how an application uses connection pool with re-configuration:

```
import oracle.jdbc.oci.*;
import oracle.jdbc.pool.*;

public class cpoolTest
{
    public static void main (String args [])
        throws SQLException
    {
        /* pass the URL and "inst1" as the database link name from tnsnames.ora */
        OracleOCIConnectionPool cpool = new OracleOCIConnectionPool
            ("scott", "tiger", "jdbc:oracle:oci8@inst1", null);
```

```
/* create virtual connection objects from the connection pool "cpool." The
   poolConfig can be null when using default values of min = 1, max = 1, and
   increment = 0, otherwise needs to set the properties mentioned earlier */
OracleOCIConnection conn1 = (OracleOCIConnection) cpool.getConnection
    ("user1", "password1");

/* create few Statement objects and work on this connection, conn1 */
Statement stmt = conn1.createStatement();

...
OracleOCIConnection conn90 = (OracleOCIConnection) cpool.getConnection
    ("user90", "password90") /* work on statement object from virtual
                             connection "conn90" */

...
/* if the throughput is less, increase the pool size */
string newmin = String.valueOf (cpool.getMinLimit());
string newmax = String.valueOf (2*cpool.getMaxLimit());
string newincr = String.valueOf (1 + cpool.getConnectionIncrement());
Properties newproperties = newProperties();
newproperties.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, newmin);
newproperties.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, newmax);
newproperties.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, newincr);
cpool.setPoolConfig (newproperties);
} /* end of main */
} /* end of cpoolTest */
```

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 ("SQL query")` is done, the statement cache is searched for a statement that matches the SQL query. If a match is found, 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`.

Statement caching in `OracleOCIConnectionPool` is a little different from the standard functionality in `OracleConnectionCacheImpl`. The

`setStmtCacheSize()` method sets the statement cache sizes of all the `OracleOCIConnection` objects retrieved from this pool. But unlike logical (`OracleConnection`) connection objects obtained from `OracleConnectionCacheImpl`, the individual cache sizes of the logical (`OracleOCIConnection`) connection objects can also be changed if desired. (The default cache size is zero.)

The following code shows the signatures of the `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
```

Types of Statement Caching used with the OCI Connection Pool

There are two forms of statement caching: implicit and explicit. (See [Chapter 13, "Statement Caching"](#) for a complete description of implicit and explicit statement caching.) Both forms of statement caching use the `setStmtCacheSize()` method. Explicit statement caching requires the JDBC application to provide a key while opening and closing `Statement` objects. Implicit statement caching does not require the JDBC application to provide the key; the caching is transparent to the application. Also in explicit statement caching, the fetch state of the result set is not cleared. So after doing a `Statement.close(key="abc")`, `Connection.prepareStatement(key="abc")` will return the `Statement` object and fetches will continue with the fetch state when the previous `Statement.close(key="abc")` is done.

For implicit statement caching, the fetch state is cleared and the cursor is re-executed, but the cursor meta data is cached to improve performance. In some cases, the client may also need to clear the meta data (through the `clearMetaData` parameter).

The following header information documents method signatures:

```
synchronized public void setStmtCacheSize (int size)
```

```
/**
 *
 * @param size Size of the Cache
 * @param clearMetaData Whether the state has to be cleared or not
 * @exception SQLException
 */
public synchronized void setStmtCacheSize (int size, boolean clearMetaData)

/**
 * Return the size of Statement Cache.
 * @return int Size of Statement Cache.
 *
 * If not set ie if statement caching is not enabled ,
 * the default 0 is returned.
 */
public synchronized int getStmtCacheSize()

/**
 * Check whether Statement
 * Caching is enabled for this pool or Not.
 */
public synchronized boolean isStmtCacheEnabled ()
```

JNDI and the OCI Connection Pool

The Java Naming and Directory Interface (JNDI) feature makes persistent the properties of Java object so 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.

Middle-Tier Authentication Through Proxy Connections

Middle-tier authentication allows one JDBC connection (session) to act as proxy for other JDBC connections. A proxy session could be required for one of the following:

- If the middle tier does not know the password of the proxy user. This is done by first authenticating using:

```
alter user jeff grant connect through scott with roles role1, role2;
```

Then the method allows you to connect as "jeff" using the already authenticated credentials of "scott". It is sometimes a security concern for the middle tier to know the passwords of all the database users. Though the created session will behave much like "jeff" was connected normally (using "jeff"/"jeff-password"), "jeff" will not have to divulge its password to the middle tier. The schema which this proxy session has access to is schema of "jeff" plus what is indicated in the list of roles. Therefore, if "scott" wants "jeff" to access its table EMP, the following code can be used:

```
create role role1;
grant select on EMP to role1;
```

The role clause can also be thought as limiting "jeff's" access to only those database objects of "scott" mentioned in the list of the roles. The list of roles can be empty.

- For accounting purposes. The transactions made via proxy sessions can be better accounted by proxying the user ("jeff"), under different users such as "scott", "scott2" assuming "scott" and "scott2" are authenticated. Transactions made under these different proxy sessions by "jeff" can be logged separately.

There are three ways to create proxy sessions in the OCI driver. Roles can be associated with any of the following options:

- **USER NAME** : This is done by supplying the user name and/or the password. The reason why the "password" option exists is so that database operations made by the user ("jeff"), can be accounted. The SQL clause is:

```
alter user jeff grant connect through scott authenticated using password;
```

Having no authenticated clause implies the default—authenticated using the user-name without the password requirement.

- **DISTINGUISHED NAME** : This is a global name in lieu of the password of the user being proxied for. So you could say "create user jeff identified globally as:

```
'CN=jeff,OU=americas,O=oracle,L=redwoodshores,ST=ca,C=us';
```

The string after the "globally as" clause is the distinguished name. It is then necessary to authenticate as:

```
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 (to be proxied) to the database. The certificate contains the distinguished encoded name. One way of generating it is by creating a wallet (using "runutl mkwallet"), then decoding the wallet to get the certificate. It is then necessary to authenticate as:

```
alter user jeff grant connect through scott authenticated using certificate;
```

The following code shows signatures of the `getProxyConnection()` method with information about the proxy type process:

```
/*
 * For creating a proxy connection. All macros are defined
 * in OracleOCIConnectionPool.java
 *
 * @param proxyType Can be one of following types
 *     PROXYTYPE_USER_NAME
 *         - This will be the normal mode of specifying the user
 *           name in proxyUser as in Oracle8i
 *
 *     PROXYTYPE_DISTINGUISHED_NAME
 *         - This will specify the distinguished name of the user
 *           in proxyUser
 *
 *     PROXYTYPE_CERTIFICATE
 *         - This will specify the proxy certificate
```

The Properties (ie prop) should be set as follows.

```
If PROXYTYPE_USER_NAME
    PROXY_USER_NAME and/or PROXY_USER_PASSWORD depending
    on how the connection-pool owner was authenticated
    to act as proxy for this proxy user
    PROXY_USER_NAME (String) = user to be proxied for
    PROXY_PASSWORD (String) = password of the user to be proxied for

else if PROXYTYPE_DISTINGUISHED_NAME
    PROXY_DISTINGUISHED_NAME (String) = (global) distinguished name of the
```

```
user to be proxied for
    else if PROXYTYPE_CERTIFICATE (byte[])
        PROXY_CERTIFICATE = certificate containing the encoded
                               distinguished name

    PROXY_ROLES (String[]) Set of roles which this proxy connection can use.
    Roles can be null, and can be associated
    with any of the above proxy methods.

    *
    * @return  connection object
    *
    * Notes: The user and password used to create OracleOCIConnectionPool()
    *        must be allowed to act as proxy for user 'us'.
    */
    public synchronized OracleConnection getProxyConnection(String proxyType,
        Properties prop)
        throws SQLException
```

For a complete code example of middle-tier authentication, see "[Middle-Tier Authentication—NtierAuth.java](#)" on page 20-93.

OCI Driver Transparent Application Failover

Transparent Application Failover (TAF) or simply *Application Failover* is a feature of the OCI driver. It enables you to automatically reconnect to a database if the database instance to which the connection is made goes down. In this case, the active transactions roll back. (A transaction rollback restores the last committed transaction.) The new database connection, though created by a different node, is identical to the original. This is true regardless of how the connection was lost.

TAF is always active and does not have to be set.

For additional details regarding OCI and TAF, see the *Programmer's Guide to the Oracle Call Interface*.

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 re-authenticated on the server-side while open cursors in the OCI application need to be re-executed.
- `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 un-successful, 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` : See above.
- `FO_EVENT_UNKNOWN` : A bad failover event.

For a complete code example that uses failover events, see "[JDBC OCI Application Failover Callbacks—OCIFailOver.java](#)" on page 20-97.

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 above possible Failover Types  
                      int event ); // One of the above 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.

OCI HeteroRM XA

Unlike the regular JDBC XA feature which works only with Oracle8i 8.1.6 and later databases, JDBC HeteroRM XA also allows you to do XA operations in Oracle8i releases prior to 8.1.6. In general, the HeteroRM XA is recommended for use whenever possible.

HeteroRM XA is enabled through the use of the `tnsEntry` and `nativeXA` properties of the `OracleXADataSource` class. [Table 14–2, "Oracle Extended Data Source Properties"](#) on page 14-5 explains these properties in detail.

For a complete discussion of XA, see [Chapter 15, "Distributed Transactions"](#).

Configuration and Installation

The Solaris shared libraries, `libheteroxa9.so` and `libheteroxa9_g.so`, enable the HeteroRM XA feature to support access to Oracle8i releases prior to release 8.1.6. The NT version of these libraries are `heteroxa9.dll` and `heteroxa9_g.dll`. In order for the HeteroRM XA feature to work properly, these libraries need to be installed and available in either the Solaris search path or the NT DLL path, depending on your system.

Note: Libraries with the `_g` suffix are debug libraries.

Exception Handling

When using the HeteroRM XA feature in distributed transactions, it is recommended that the application simply check for `XAException` or `SQLException`, rather than `OracleXAException` or `OracleSQLException`.

See "[HeteroRM XA Messages](#)" on page B-12 for a listing of HeteroRM XA messages.

Note: The mapping from SQL error codes to standard XA error codes does not apply to the HeteroRM XA feature.

HeteroRM XA Code Example

The following portion of code shows how to enable the HeteroRM XA feature. For a complete code example of the HeteroRM XA feature, see "[HeteroRM XA—XA6.java](#)" on page 20-116.

```
// Create a XADataSource instance
```

```
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL(url);

// Set the nativeXA property to use HeteroRM XA feature
oxds.setNativeXA(true);

// Set the tnsEntry property to an older DB as required
oxds.setTNSEntryName("ora805");
```

Accessing PL/SQL Index-by Tables

The Oracle JDBC OCI driver enables JDBC applications to make PL/SQL calls with index-by table parameters.

See "[Accessing a PL/SQL Index-by Table from JDBC—PLSQLIndexTab.java](#)" on page 20-11 for a complete code example that uses PL/SQL index-by table parameters.

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

Overview

The Oracle JDBC OCI driver supports PL/SQL index-by tables of scalar datatypes. [Table 16-1](#) displays the supported scalar types and the corresponding JDBC typecodes.

Table 16-1 *PL/SQL Types and Corresponding JDBC Types*

PL/SQL Types	JDBC Types
BINARY_INTEGER	NUMERIC
NATURAL	NUMERIC
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` (including function return values), or `IN OUT` parameters. For general information about PL/SQL syntax, see the *PL/SQL User's Guide and Reference*.

The following sections describe the methods used to bind and register PL/SQL index-by tables.

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 16-2](#) describes the arguments of the `setPlsqlIndexTable()` method.

Table 16-2 Arguments of the `setPlsqlIndexTable()` Method

Argument	Description
<code>int paramIndex</code>	This argument indicates the parameter position within the statement.
<code>Object arrayData</code>	This argument is an array of values to be bound to the PL/SQL index-by table parameter. The value is of type <code>java.lang.Object</code> , and the value can be a Java primitive type array such as <code>int[]</code> or a Java object array such as <code>BigDecimal[]</code> .
<code>int maxLen</code>	This argument specifies the maximum table length of the index-by table bind value which defines the maximum possible <code>curLen</code> for batch updates. For standalone binds, <code>maxLen</code> should use the same value as <code>curLen</code> . This argument is required.

Table 16–2 Arguments of the `setPlsqlIndexTable ()` Method (Cont.)

Argument	Description
<code>int curLen</code>	This argument specifies the actual size of the index-by table bind value in <code>arrayData</code> . If the <code>curLen</code> value is smaller than the size of <code>arrayData</code> , only the <code>curLen</code> number of table elements is passed to the database. If the <code>curLen</code> value is larger than the size of <code>arrayData</code> , the entire <code>arrayData</code> is sent to the database.
<code>int elemSqlType</code>	This argument specifies the index-by table element type based on the values defined in the <code>OracleTypes</code> class.
<code>int elemMaxLen</code>	This argument specifies the index-table element maximum length in case the element type is <code>CHAR</code> , <code>VARCHAR</code> , or <code>RAW</code> . 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 this section describes 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 registerIndexTableOutParameter  
    (int paramIndex, int maxlen, int elemSqlType, int elemMaxLen)  
    throws SQLException
```

[Table 16–3](#) describes the arguments of the `registerIndexTableOutParameter()` method.

Table 16–3 Arguments of the registerIndexTableOutParameter () Method

Argument	Description
<code>int paramIndex</code>	This argument indicates the parameter position within the statement.
<code>int maxlen</code>	This argument specifies the maximum table length of the index-by table bind value to be returned.
<code>int elemSqlType</code>	This argument specifies the index-by table element type based on the values defined in the <code>OracleTypes</code> class.
<code>int elemMaxLen</code>	This argument specifies the index-by 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 `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 RAW. 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	<code>getPlsqlIndexTable(int)</code>
Oracle mappings	<code>getOraclePlsqlIndexTable(int)</code>
Java primitive type mappings	<code>getPlsqlIndexTable(int, Class)</code>

JDBC Default Mappings The `getPlsqlIndexTable()` method with the `(int)` signature returns index-by table elements using JDBC default mappings.

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

Table 16–4 describes the argument of the `getPlsqlIndexTable()` method.

Table 16–4 Argument of the `getPlsqlIndexTable ()` Method

Argument	Description
<code>int paramIndex</code>	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 typecode, 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 a `BigDecimal[]` array to access the table element values. (See ["Datatype Mappings"](#) on page 3-16 for a list of default mappings.)

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 16–5](#) describes the argument of the `getOraclePlsqlIndexTable()` method.

Table 16–5 Argument of the `getOraclePlsqlIndexTable ()` Method

Argument	Description
<code>int paramIndex</code>	This argument indicates the parameter position within the statement.

The return value is an `oracle.sql.Datum` array and the elements in the `Datum` array will be 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. (The code for registration is omitted.)

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

...

// execute 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()` method with the `(int, Class)` signature returns index-by table elements in Java primitive types. The return value is a Java array.

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

[Table 16–6](#) describes the arguments of the `getPlsqlIndexTable()` method.

Table 16–6 Arguments of the `getPlsqlIndexTable()` Method

Argument	Description
<code>int paramIndex</code>	This argument indicates the parameter position within the statement.

Table 16–6 Arguments of the `getPlsqlIndexTable ()` Method

Argument	Description
Class <code>primitiveType</code>	<p>This argument specifies a Java primitive type to which the index-by table elements are to be converted. For example, if you specify <code>java.lang.Integer.TYPE</code>, the return value is an <code>int</code> array.</p> <p>The following are the possible values of this parameter:</p> <ul style="list-style-type: none"> <code>java.lang.Integer.TYPE</code> <code>java.lang.Long.TYPE</code> <code>java.lang.Float.TYPE</code> <code>java.lang.Double.TYPE</code> <code>java.lang.Short.TYPE</code>

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`, so the return value of the `getPlsqlIndexTable ()` method is an `int` array.

```

OracleCallableStatement funcnone = (OracleCallableStatement)
    conn.prepareStatement ("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.getPsqlIndexTable (1, java.lang.Integer.TYPE);  
  
// print the elements  
for (int i=0; i<values.length; i++)  
    System.out.println (values[i]);
```

Java Transaction API

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, which is described fully in this chapter. The process of including database SQL updates within a transaction is referred to as enlisting a database resource.

This chapter assumes that you have a working knowledge of JTA. The discussion focuses mostly on examples and explaining the differences between Sun Microsystems's JTA specification and the Oracle JTA implementation. See <http://www.javasoft.com> for the Sun Microsystems JTA specification.

- [Transaction Overview](#)
- [JTA Summary](#)
- [Binding Transactional Objects in the Namespace](#)
- [JTA Client-Side Demarcation](#)
- [JTA Server-Side Database Enlistment](#)
- [Configuring Two-Phase Commit Engine](#)
- [Creating DataSource Objects Dynamically](#)
- [JDBC Restrictions](#)

Transaction Overview

Transactions manage changes to multiple databases within a single application as a unit of work. That is, if you have an application that manages data within one or more databases, you can ensure that all changes in all databases are committed at the same time if they are managed within a transaction.

Transactions are described in terms of ACID properties, which are as follows:

- *Atomic*: all changes to the database made in a transaction are rolled back if any change fails.
- *Consistent*: the effects of a transaction take the database from one consistent state to another consistent state.
- *Isolated*: the intermediate steps in a transaction are not visible to other users of the database.
- *Durable*: when a transaction is completed (committed or rolled back), its effects persist in the database.

The JTA implementation, specified by Sun Microsystems, relies heavily on the JDBC 2.0 specification and XA architecture. The result is a complex requirement on applications in order to ensure that the transaction is managed completely across all databases. Sun Microsystems's specifies Java Transaction API (JTA) 1.0.1 and JDBC 2.0 on <http://www.javasoft.com>.

You should be aware of the following when using JTA within the Oracle9i environment:

- [Global and Local Transactions](#)
- [Demarcating Transactions](#)
- [Enlisting Resources](#)
- [Two-Phase Commit](#)
- [Enlisting Resources](#)
- [JTA Limitations](#)

Global and Local Transactions

Whenever your application connected to a database using JDBC or a SQL server, you were creating a transaction. However, the transaction involved only the single database and all updates made to the database were committed at the end of these changes. This is referred to as a local transaction.

A global transaction involves a complicated set of management objects—objects that track all of the objects and databases involved in the transaction. These global transaction objects—`TransactionManager` and `Transaction`—track all objects and resources involved in the global transaction. At the end of the transaction, the `TransactionManager` and `Transaction` objects ensure that all database changes are atomically committed at the same time.

Within a global transaction, you cannot execute a local transaction. If you try, the following error will be thrown:

```
ORA-2089 "COMMIT is not allowed in a subordinate session."
```

Some SQL commands implicitly execute a local transaction. All SQL DDL statements, such as "CREATE TABLE", implicitly starts and commits a local transaction under the covers. If you are involved in a global transaction that has enlisted the database that the DDL statement is executing against, the global transaction will fail.

Demarcating Transactions

A transaction is said to be demarcated, which means that each transaction has a definite beginning and end point. For example, in an interactive tool such as SQL*Plus, a transaction is started with the SQL `BEGIN` statements. Then, each SQL DML statement is part of the transaction. A transaction ends when a SQL `COMMIT` or `ROLLBACK` statement is issued.

For JTA, transactions are demarcated programmatically through the `UserTransaction` object, which must be bound within the namespace. The client or server starts the transaction by invoking the `UserTransaction.begin` method; it ends the transaction by invoking either the `UserTransaction.commit` or `UserTransaction.rollback` methods.

The originating client or object that starts the transaction must also end the transaction with a commit or rollback. If the client begins the transaction, calls out to a server object, the client must end the transaction after the invoked method returns. The invoked server object cannot end the transaction.

UserTransaction Interface

The following are the methods that you can use for transaction demarcation. These methods are defined within the `javax.transaction.UserTransaction` interface:

```
public abstract void begin() throws NotSupportedException, SystemException;
```

Creates a new transaction and associates the transaction with the thread.

Exceptions:

- `NotSupportedException`: Thrown if the thread is already involved with a transaction. Nested transactions are not supported.
- `SystemException`: Thrown if an unexpected error condition occurs.

```
public abstract void commit() throws RollbackException, HeuristicMixedException,  
HeuristicRollbackException, SecurityException, IllegalStateException,  
SystemException;
```

Completes the existing transaction by saving all changes to resources involved in the transaction. The thread is disassociated from this transaction when this method finishes.

Exceptions:

- `RollbackException`: Thrown if any resource within the transaction could not commit successfully. All resource changes are rolled back.
- `HeuristicMixedException`: Thrown to indicate that some of the resources were committed; some were rolled back.
- `HeuristicRollbackException`: Thrown to indicate that some updates to resources involved in the transaction were rolled back.
- `SecurityException`: Thrown when the thread is not allowed to commit the transaction based on a security violation.
- `IllegalStateException`: Thrown if the current thread has not been associated with a transaction. This occurs if you try to commit a transaction that was never started.
- `SystemException`: Thrown if an unexpected error condition occurs.

```
public abstract void rollback() throws IllegalStateException, SecurityException,  
SystemException;
```

Roll back the transaction associated with the current thread.

Exceptions:

- `SecurityException`: Thrown when the thread is not allowed to roll back the transaction based on a security violation.
- `IllegalStateException`: Thrown if the current thread has not been associated with a transaction. This occurs if you try to roll back a transaction that was never started.

- `SystemException`: Thrown if an unexpected error condition occurs.

```
public abstract int getStatus() throws SystemException;
```

Retrieve the transaction status associated with the current thread.

Exceptions:

- `SystemException`: Thrown if an unexpected error condition occurs.

```
public abstract void setRollbackOnly() throws IllegalStateException,  
    SystemException;
```

Modify the transaction associated with the current thread so that the outcome results in a rollback.

Exceptions:

- `IllegalStateException`: Thrown if the current thread has not been associated with a transaction. This occurs if you try to set for a roll back a transaction that was never started.
- `SystemException`: Thrown if an unexpected error condition occurs.

```
public abstract void setTransactionTimeout(int seconds) throws SystemException;
```

Set the timeout value in seconds for the transaction associated with this current thread.

Exceptions:

- `SystemException`: Thrown if an unexpected error condition occurs.

Enlisting Resources

Each resource, such as an Oracle database, that you want managed in the global transaction must be enlisted in that transaction. The transaction manager tracks all resources involved in the global transaction.

While there are several methods for retrieving a JDBC connection to a database, only one of these methods causes the database to be included in a JTA transaction. The following table lists the normal methods for retrieving JDBC connections:

Table 17–1 JDBC Methods

Retrieval Method	Description
<code>OracleDriver().defaultConnection()</code>	Pre-JDBC 2.0 method for retrieving the local connection. Use only within local transactions.
<code>DriverManager.getConnection("jdbc:oracle:kprb: ")</code>	Pre-JDBC 2.0 method for retrieving the local connection. Use only within local transactions.
<code>DataSource.getConnection("jdbc:oracle:kprb: ")</code>	JDBC 2.0 method for retrieving connections to the local databases. Can be used for JTA transactions.

Of these methods, only the `DataSource` object can be used to include a database resource in the global transaction. You must bind any `DataSource` object to be used within a JTA transaction with the `-jta` option of the `bindds` command. This binds an Oracle-specific JTA `DataSource` within the namespace. You cannot use any other type of `DataSource` for global transactions.

In order to ensure that the statements are included within a global transaction, you must do the following:

1. Bind a JTA `DataSource` object (`OracleJTADDataSource`) in the JNDI namespace. There are several types of `DataSource` objects that you can bind. You must bind the JTA type in order for this database to be included in the global transaction. You must retrieve the connection within the context of a global transaction; that is, retrieve the connection after the `begin` method of the `UserTransaction` object.
2. The method must retrieve the `DataSource` object from the JNDI namespace after the global transaction has started.
3. Retrieve the connection object from this `DataSource` object using the `getConnection` method.

An object—client or server object—opens a JDBC connection to a remote database through the `DataSource` object within the context of a global transaction. The client can open connections to remote databases through any client JDBC driver. Server objects can only open connections to databases through either a JDBC KPRB driver or a JDBC server-side Thin driver.

If your transaction involves more than one database, you must specify an Oracle9i database as the two-phase commit engine. See "[Configuring Two-Phase Commit Engine](#)" on page 17-21 for more information.

Note: At this time, the Oracle JTA implementation does not support including non-Oracle databases in a global transaction.

Two-Phase Commit

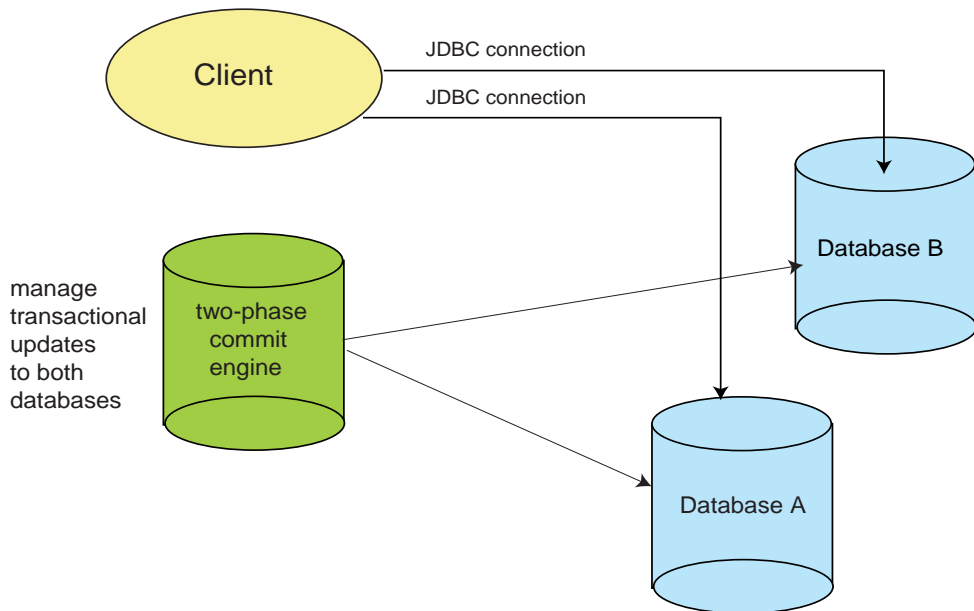
One of the primary advantages for a global transaction is the number database resources managed as a single unit within the transaction. If your global transaction involves more than one database resource, you must specify a two-phase commit engine, which is an Oracle9i database designated to manage the changes to all databases within the transaction. The two-phase commit engine is responsible for ensuring that when the transaction ends, all changes to all databases are either totally committed or fully rolled back.

On the other hand, if your global transaction has multiple server objects, but only a single database resource, you do not need to specify a two-phase commit engine. The two-phase commit engine is required only to synchronize the changes for multiple databases. If you have only a single database, single-phase commit can be performed by the transaction manager.

Note: Your two-phase commit engine can be any Oracle9i database. It can be the database where your server object exists, or even a database that is not involved in the transaction at all. See ["Configuring Two-Phase Commit Engine"](#) on page 17-21 for a full explanation of the two-phase commit engine setup.

[Figure 17-1](#) shows two databases enlisted in the transaction and another database that is designated as the two-phase commit engine. Databases A and B are enlisted when a JDBC connection is opened, within a global transaction, to each database. The two-phase commit engine must be identified within the `UserTransaction` object and database links from the two-phase commit engine to each database involved in the transaction must be created. See ["Configuring Two-Phase Commit Engine"](#) on page 17-21 on how to perform this setup.

When the global transaction ends, the two-phase commit engine ensures that all changes made to the databases A and B are committed or rolled back simultaneously.

Figure 17-1 Two-Phase Commit for Global Transactions

JTA Limitations

The following are the portions of the JTA specification that Oracle9i does not support.

Nested Transactions

Nested transactions are not supported in this release. If you attempt to begin a new transaction before committing or rolling back any existing transaction, the transaction service throws a `SubtransactionsUnavailable` exception.

Interoperability

The transaction services supplied with this release do not interoperate with other JTA implementations.

Timeouts

Oracle only supports a database resource idle timeout through the `UserTransaction` object. Thus, Oracle does not recommend that you use timeouts.

JTA Summary

The following sections summarize the details for demarcating the transaction and enlisting the database in the transaction. These details are explained and demonstrated in the rest of the chapter. However, these tables provide a reference point for you.

Environment Initialization

Before you can retrieve the `UserTransaction` or `DataSource` bound objects from the JNDI namespace, you must provide the following before the JNDI lookup:

- authentication information, such as username and password
- namespace URL

Table 17–2 Environment Setup For Transactional Object Retrieval

Source	Qualifiers	Environment Setup
		<i>Setup includes authentication information, and namespace URL</i>
Client	Retrieves a remote object or a remote database connection.	Must always provide the environment setup before retrieving the <code>UserTransaction</code> from a remote JNDI provider. All JNDI providers are remote from a true client.
Server	<ul style="list-style-type: none"> ■ Can use in-session activation to retrieve a local object or local database connection. 	If the JNDI provider is within the same database as the object, it can use in-session lookup. Since the server uses its own session for the lookup, no setup is required.
	<ul style="list-style-type: none"> ■ Retrieves a remote object or a remote database connection. 	The JNDI provider is remote, so the server object must always provide the environment setup.

Methods for Enlisting Database Resources

The `DataSource` object is used to explicitly enlist the database in the JTA transaction. In order for the database to be correctly enlisted, the `DataSource` must be bound correctly, and the retrieval mechanism can be one of three methods. These are discussed below:

Table 17–3 JDBC 2.0 DataSource Overview

JDBC 2.0 DataSource	
Binding	You must bind a JTA DataSource into the namespace with the <code>bindds</code> command. The <code>bindds</code> command must contain the <code>-dstype jta</code> option.
Retrieving DataSource object from remote JNDI provider	<ol style="list-style-type: none"> 1. Provide the environment <code>Hashtable</code>, which contains authentication information and namespace URL. 2. Retrieve the <code>DataSource</code> object through a JNDI lookup that contains the <code>"jdbc_access://"</code> prefix.
Retrieving DataSource object from local JNDI provider	Retrieve the <code>DataSource</code> object using in-session activation. Environment setup and <code>"jdbc_access://"</code> prefix is not required.

Summary of Single-Phase and Two-Phase Commit

[Table 17–4](#) summarizes the single-phase commit scenario. It covers the JNDI binding requirements and the application implementation runtime requirements.

Table 17–4 Single-Phase Commit

Aspect	Description
Binding	<ul style="list-style-type: none"> ■ No binding required for <code>UserTransaction</code>. The <code>UserTransaction</code> object is created for you. ■ Bind a <code>DataSource</code> object using the <code>bindds</code> command.
Runtime	<ul style="list-style-type: none"> ■ Retrieve the <code>UserTransaction</code> either through a JNDI lookup. ■ Your runtime is responsible for starting and terminating the transaction. ■ If using the <code>DataSource</code> object to manage SQL DML statements within the transaction, retrieve the <code>DataSource</code>.

[Table 17–5](#) summarizes the two-phase commit scenario.

Table 17–5 Two-Phase Commit Requirements

Aspect	Requirements
Binding <code>UserTransaction</code> One of two scenarios:	<p>Scenario one is where you bind the <code>UserTransaction</code> WITH a username and password that is to be used to complete all global transactions started from this <code>UserTransaction</code>.</p> <ul style="list-style-type: none"> ■ You bind a <code>UserTransaction</code> object with the fully-qualified database address of the two-phase commit engine and its username and password. ■ You bind <code>DataSource</code> objects for each database involved in the transaction with a fully-qualified public database link from the two-phase commit engine to itself. <hr/> <p>Scenario two is where you bind the <code>UserTransaction</code> WITHOUT a username and password. Thus, the username that is used when retrieving the <code>UserTransaction</code> is the user that completes the transaction.</p> <ul style="list-style-type: none"> ■ You bind a <code>UserTransaction</code> object with the fully-qualified database address of the two-phase commit engine. ■ You bind <code>DataSource</code> objects for each database involved in the transaction with a fully-qualified public database link from the two-phase commit engine to itself.
Binding <code>DataSource</code>	You must bind a JTA <code>DataSource</code> for each database involved in the transaction. You must create public database links, as discussed in the System Administration section.
System Administration	<ul style="list-style-type: none"> ■ The user that completes the transaction (as described in the binding section) must have the privilege to commit the transaction on all included databases. There are one of two methods for ensuring that the user can complete the transaction. <ul style="list-style-type: none"> - If the username is not bound with the <code>UserTransaction</code> object, the user that retrieves the <code>UserTransaction</code> both starts and stops the transaction. Thus, this user must be created on all involved database in order to be able to open a session to all databases. - If the username is bound with the <code>UserTransaction</code> object is different than the user that retrieves the <code>UserTransaction</code> object, the username bound with the <code>UserTransaction</code> object must be given explicit privilege to complete a transaction it did not start. Thus, make sure that this user exists on each database in order to open sessions to all databases and grant it the "CONNECT, REMOVE, CREATE SESSION, and FORCE ANY TRANSACTION" privileges on each database. ■ Create public database links from the two-phase commit engine to each database involved.
Runtime	Runtime requirements are the same as indicated in the single-phase commit table.

Binding Transactional Objects in the Namespace

For most global transactions, you will need to bind at least one of the following objects in the namespace:

- `UserTransaction` object
- `DataSource` object

Bind `UserTransaction` Object in the Namespace

The `bindut` command binds a `UserTransaction` object in the namespace. This object is used for demarcation of global transactions.

You must bind a `UserTransaction` object for both single and two-phase commit transactions through the `bindut` command of the `sess_sh` tool.

The options used to bind a `UserTransaction` object depend on whether the transaction uses a single or two-phase commit, as described below:

Single-Phase Commit Binding for `UserTransaction` Single-phase commit requires the JNDI bound name for the `UserTransaction` object. You do not need to provide the address to a two-phase commit engine. For example, the following binds a `UserTransaction` with the name of `"/test/myUT"` that exists for a single-phase commit transaction:

```
bindut /test/myUT
```

To bind a `UserTransaction` object to the name `"/test/myUT"` in the namespace located on `nsHost` through the `sess_sh` command, execute the following:

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER  
& bindut /test/myUT
```

Two-Phase Commit Binding for `UserTransaction` Two-phase commit binding requires the JNDI bound name for the `UserTransaction` object and the address to a two-phase commit engine. You provide a URL for the two-phase commit engine in the `bindut` command, which can be either a JDBC URL or a `sess_iiop` URL.

Note: The client needs the same information to retrieve the `UserTransaction` as you give within the `bindut` command.

In addition, you can bind a username and password with the `UserTransaction` object.

- If you do not bind a username and password with the `UserTransaction`, the user that retrieved the `UserTransaction` will be the same user that is used to

perform the commit or rollback for the two-phase commit on all involved databases.

- If you bind a username and password with the `UserTransaction`, then this is the username that the two-phase commit will be committed or rolled back with on all involved databases. The transaction will be started by the user that retrieves the `UserTransaction` object; it will be completed by the user bound with the `UserTransaction` object.

The username that is used to commit or rollback the two-phase commit transaction must be created on the two-phase commit engine and on each database involved in the transaction. It needs to be created so that it can open a session from the two-phase commit engine to each of the involved databases using database links. Secondly, it must be granted the `CONNECT`, `RESOURCE`, `CREATE SESSION` privileges to be able to connect to each of these databases. For example, if the user that is needed for completing the transaction is `SCOTT`, you would do the following on the two-phase commit engine and each database involved in the transaction:

```
CONNECT SYSTEM/MANAGER;
CREATE USER SCOTT IDENTIFIED BY SCOTT;
GRANT CONNECT, RESOURCE, CREATE SESSION TO SCOTT;
```

Lastly, if you bound a username and password with the `UserTransaction` object, it will be using a different username to finalize the transaction than the username used to start the transaction. For this to be allowed, you must grant the `FORCE ANY TRANSACTION` privileges on each database involved in the transaction in order for two separate users to start and stop the transaction. If `SCOTT` is the username bound with the `UserTransaction` object, you would need to do the following in addition to the previous grant:

```
GRANT FORCE ANY TRANSACTION TO SCOTT;
```

The following binds a `UserTransaction` with the name of `"/test/myUT"` and a two-phase commit engine at `"2pcHost"` using a JDBC URL:

```
bindut /test/myUT -url jdbc:oracle:thin:@2pcHost:5521:ORCL
```

To bind the `UserTransaction` in the namespace designating the two-phase commit engine at `dbsun.mycompany.com` with a `sess_iiop` URL:

```
bindut /test/myUT -url sess_iiop://dbsun.mycompany.com:2481:ORCL
```

When the transaction commits, the `UserTransaction` communicates with the two-phase engine designated in the `-url` option to commit all changes to all included databases. In this example, the username and password were not bound

with the `UserTransaction` object, so the user that retrieves the `UserTransaction` object from the JNDI namespace is used to start and stop the transaction. Thus, this user must exist on all involved databases and the two-phase commit engine. The `UserTransaction` tracks all databases involved in the transaction; the two-phase commit engine uses the database links for these databases to complete the transaction.

Note: If you change the two-phase commit engine, you must update all database links on all `DataSource` objects involved in the transaction, and rebind the `UserTransaction`.

Bind `DataSource` Object in the Namespace

The `bindds` command binds a `DataSource` object in the JNDI namespace. In order to enlist any database in a global transaction—including the local database—you must bind a JTA `DataSource` object to identify each database included in the transaction. There are multiple types of `DataSource` objects for use with certain scenarios. However, for use with JTA transactions, you must bind a JTA `DataSource` object, also known as an `OracleJTADDataSource` object, to identify each database included in the transaction. See the `bindds` command of the `sess_sh` tool in the *Oracle9i Java Tools Reference* for a description of other `DataSource` object types.

Single-Phase Commit Scenario In a single-phase commit scenario, the transaction only includes a single database in the transaction. Since no coordination for updates to multiple databases is needed, you do not need to specify a coordinator. Instead, you simply provide the JNDI bound name and the URL address information for this database within the `OracleJTADDataSource` object. You do not need to provide a database link for a transaction coordinator.

Use the `bindds` command of the `sess_sh` tool to bind an `DataSource` object in the namespace. The full command is detailed in the *Oracle9i Java Tools Reference*. For example, the following binds an `OracleJTADDataSource` with the name of `"/test/empDS"` that exists within a single-phase commit transaction with the `bindds` command:

```
bindds /test/empDS -url jdbc:oracle:thin:@empHost:5521:ORCL -dstype jta
```

After binding the `DataSource` object in the namespace, the server can enlist the database within a global transaction.

Two-Phase Commit Scenario If multiple databases are to be included in the global transaction, you will need a two-phase commit engine, which is an Oracle9i database that is configured to be the transaction coordinator. Basically, the two-phase commit engine must have database links to each of the databases involved in the transaction. When the transaction ends, the transaction manager notifies the two-phase commit engine to either coordinate the commit of all changes to all involved databases or coordinate a roll back of these same changes.

In order to facilitate this coordination, you must configure the following:

1. Your system administrator must create fully-qualified public database links from the two-phase commit engine (Oracle9i database) to each database involved in the transaction. These database link names must be included when binding the `OracleJTADataSource` object.
2. Bind a JTA `DataSource` (`OracleJTADataSource`) object for each database in the transaction. You must include the following in the `bindds` command:
 - a. The JNDI bound name for the object
 - b. The URL for creating a connection to the database
 - c. The fully-qualified public database link from the two-phase commit engine to this database

Note: In a two-phase commit scenario, the `DataSource` object is bound, with respect to the two-phase commit engine. If you change the two-phase commit engine, you must update all database links, and rebind all concerned `DataSource` and `UserTransaction` objects.

The following example binds the `empDS` JTA `DataSource` into the namespace with `2pcToEmp` as the database link name created on the two-phase commit engine:

```
% bindds /test/empDS -url jdbc:oracle:thin:@dbsun:5521:ORCL
                        -dstype jta -dblink 2pcToEmp.oracle.com
```

JTA Client-Side Demarcation

For JTA, client-side demarcated transactions are programmatically demarcated through the `UserTransaction` interface (see "[UserTransaction Interface](#)" on page 17-3). A `UserTransaction` object must be bound with the `bindut`

command into the namespace (see ["Bind UserTransaction Object in the Namespace"](#) on page 17-12). With client-side transaction demarcation, the client controls the transaction. The client starts a global transaction by invoking the `UserTransaction.begin` method; it ends the transaction by invoking either the `commit` or `rollback` methods. In addition, the client must always set up an environment including a `Hashtable` with authentication information and namespace location URL.

To include a remote database within the transaction from a client, you must use a `DataSource` object, which has been bound in the namespace as a JTA `DataSource`. Then, invoke the `getConnection` method of the `DataSource` object after the transaction has started, and the database is included in the global transaction. See ["Enlisting Resources"](#) on page 17-5 for more information.

The following must occur in the client runtime to demarcate the transaction:

1. Initialize a `Hashtable` environment with the namespace address and authentication information.
2. Retrieve the `UserTransaction` object from the namespace within the client logic. When you retrieve the `UserTransaction` object from the client, the URL must consist of `"jdbc_access://"` prefix before the JNDI name.
3. Start the global transaction within the client using `UserTransaction.begin()`.
4. Enlist any database resources to be included in the transaction by opening a connection to the specified database, as follows:
 - a. Retrieve the `DataSource` object from the namespace within the client logic. When you retrieve the `DataSource` object from any client, the URL must consist of `"jdbc_access://"` prefix before the JNDI name.
 - b. Open a connection to the database through `DataSource.getConnection` method.
5. Retrieve the object reference.
6. Invoke any object methods to be included in the transaction.
7. Invoke SQL DML statements against any enlisted databases. SQL DDL creates a local transaction that will abort the global transaction. Thus, SQL DDL cannot be executed within a JTA transaction.
8. End the transaction through `UserTransaction.commit()` or `UserTransaction.rollback()`.

[Example 17-1](#) shows a client that invokes a server object and enlists a single database within the transaction.

Example 17-1 Employee Client Code for Client Demarcated Transaction

Before starting the client, you must first bind the `UserTransaction` and `DataSource` objects in the JNDI namespace. See ["Bind UserTransaction Object in the Namespace"](#) on page 17-12 and ["Bind DataSource Object in the Namespace"](#) on page 17-14 for directions on the binding these objects.

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER
> bindut /test/myUT
> bindds /test/DataSource/empDB -url jdbc:oracle:thin:@empHost:5521:ORCL
    -dstype jta
```

Developing the Client Application

```
//Set up the service URL to where the UserTransaction object
//is bound. Since from the client, the connection to the database
//where the namespace is located can be communicated with over either
//a Thin or OCI JDBC driver. This example uses a Thin JDBC driver.
String namespaceURL = "jdbc:oracle:thin:@nsHost:1521:ORCL";

// lookup usertransaction object in the namespace
//1.(a) Authenticate to the database.
// create InitialContext and initialize for authenticating client
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put (Context.SECURITY_PRINCIPAL, "SCOTT");
env.put (Context.SECURITY_CREDENTIALS, "TIGER");
env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
//1.(b) Specify the location of the namespace where the transaction objects
// are bound.
env.put(jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
Context ic = new InitialContext (env);

//2. Retrieve the UserTransaction object from JNDI namespace
ut = (UserTransaction)ic.lookup ("jdbc_access://test/myUT");

//3. Start the transaction
ut.begin();

//4.(a) Retrieve the DataSource (that was previously bound with bindds in
// the namespace. After retrieving the DataSource...
```

```
// get a connection to a database. You need to provide authentication info
// for a remote database lookup, similar to what you would do from a client.
// In addition, if this was a two-phase commit transaction, you must provide
// the username and password.
DataSource ds = (DataSource)ic.lookup ("jdbc_access://test/empDB");

//4.(b). Get connection to the database through DataSource.getConnection
// in this case, the database requires the same username and password as
// set in the environment.
Connection conn = ds.getConnection ("SCOTT", "TIGER");

//5. Retrieve the server object reference
// lookup employee object in the namespace
Employee employee = (Employee)ic.lookup (sessiopURL + objectName);

//6. Perform business logic.
...

//7. Close the database connection.
conn.close ();

//8. End the transaction
//Commit the updated value
ut.commit ();
}
```

JTA Server-Side Database Enlistment

If the database where the server is located also contains the namespace, you can do an in-session lookup. That is, you do not need to provide any environment setup. However, if the server is located on a separate machine than the namespace, you must initialize the environment exactly as specified in ["JTA Client-Side Demarcation"](#) on page 17-15. This section assumes that the namespace and the server object exist on the same machine and demonstrates an in-session lookup of the `UserTransaction` and `DataSource` objects.

You do the following for a server to demarcate the transaction and enlist a database within the transaction:

1. The system administrator binds the required JTA objects in the namespace.
 - a. Bind the `UserTransaction` object within the namespace.
 - b. Bind the `DataSource` object within the namespace.

Note: If using more than one database, you will need to setup for a two-phase commit. See "[Configuring Two-Phase Commit Engine](#)" on page 17-21 for more information.

2. Retrieve the `UserTransaction` object from the namespace using in-session activate. No environment is required and no URL prefix is required. The lookup requires only the JNDI name of the bound object.
3. Start the global transaction within the client using `UserTransaction.begin()`.
4. Enlist any database resources to be included in the transaction by opening a JDBC connection to the specified database, as follows:
 - a. Retrieve the `DataSource` object from the namespace using in-session activate. No URL prefix is required. The lookup requires only the JNDI name of the bound object.
 - b. Open a connection to the database through the `DataSource.getConnection()`. If the username and password that was used to authenticate to this server is the correct authentication information, provide no arguments. However, if the authentication requires a different username and password, provide these as input parameters to the `getConnection()` method call.
5. Invoke SQL DML statements against any enlisted databases.
6. Close any open database connections.
7. End the transaction through `UserTransaction.commit()` or `UserTransaction.rollback()`.

Example 17-2 Enlist Database in Single Phase Transaction

The following example shows how a server object can enlist the local database in the transaction. The global transaction is initialized, the connection is retrieved through the `DataSource.getConnection()` method, and statements are executed against the local database. These statements are committed when the global transaction is committed.

Since the `UserTransaction` and `DataSource` objects are bound in the namespace local to this host, you can perform an in-session activation. This means that no environment information is necessary (unless performing a two-phase

commit scenario) and the lookup uses only the JNDI name, without the "jdbc_
access://" prefix.

```
//retrieve the initial context.
InitialContext ic = new InitialContext ();

// lookup the usertransaction
UserTransaction ut = (UserTransaction)ic.lookup ("/test/myUT");

//start the transaction
ut.begin ();

// get a connection to the local database. If this was a two-phase commit
// transaction, you would provide the username and password for the 2pc engine
DataSource ds = (DataSource)ic.lookup ("/test/empDS");

// get connection to the local database through DataSource.getConnection
Connection conn = ds.getConnection ("SCOTT", "TIGER");

//perform your SQL against the database.
//prepare and execute a sql statement. retrieve the employee's selected benefits
PreparedStatement ps =
    conn.prepareStatement ("update emp set ename = :(employee.name),
        sal = :(employee.salary) where empno = :(employee.number)");
    .... //do work
ps.close();
}

//close the connection
conn.close();

// commit the transaction
ut.commit ();

//return the employee information.
return new EmployeeInfo (name, empno, (float)salary);
```

Example 17-3 Using SQLJ with Explicit Enlistment

As in [Example 17-2](#), you would retrieve the JTA DataSource from the JNDI provider, retrieve the connection, retrieve a context from that connection, and then provide the context on the SQLJ command-line.

```
//retrieve the initial context.
InitialContext ic = new InitialContext ();
```



```

// lookup the usertransaction
UserTransaction ut = (UserTransaction)ic.lookup ("/test/myUT");

//start the transaction
ut.begin ();

// get a connection to the local database. If this was a two-phase commit
// transaction, you would provide the username and password for the 2pc engine
DataSource ds = (DataSource)ic.lookup ("/test/empDS");

// get connection to the local database through DataSource.getConnection
Connection conn = ds.getConnection ("SCOTT", "TIGER");

//setup the context for issuing SQLJ against the database
DefaultContext defCtx = new DefaultContext (conn);

//issue SQL DML statemetns against the database
#sql [defCtx] { update emp set ename = :(remoteEmployee.name),
                sal = :(remoteEmployee.salary)
                where empno = :(remoteEmployee.number) };

//close the connection
conn.close();

// commit the transaction
ut.commit ();

//return the employee information.
return new EmployeeInfo (name, empno, (float)salary);

```

Configuring Two-Phase Commit Engine

When multiple databases are included in a global transaction, the changes to these resources must all be committed or rolled back at the same time. That is, when the transaction ends, the transaction manager contacts a coordinator—also known as a two-phase commit engine—to either commit or roll back all changes to all included databases. The two-phase commit engine is an Oracle9i database that is configured with the following:

- Fully-qualified database links from the itself to each of the databases involved in the transaction. When the transaction ends, the two-phase commit engine communicates with the included databases over their fully-qualified database links.

- A user that is designated to create sessions to each database involved and is given the responsibility of performing the commit or rollback. The user that does the communication must be created on all involved databases and be given the appropriate privileges.

In order to facilitate this coordination, you must configure the following:

1. Designate an Oracle9i database as the two-phase commit engine.
2. Configure fully-qualified public database links (using the `CREATE DATABASE LINK` command) from the two-phase commit engine to each database that may be involved in the global transaction. This is necessary for the two-phase commit engine to communicate with each database at the end of the transaction. These database link names must be included when binding the JTA `DataSource` (`OracleJTADDataSource`) object.
3. Bind a JTA `DataSource` (`OracleJTADDataSource`) object for each database in the transaction. You must include the following in the `bindds` command:
 - a. The JNDI bound name for the object
 - b. The URL for creating a connection to the database
 - c. The fully-qualified database link from the two-phase commit engine to this database

Provide the fully-qualified database link name in the `-dblink` option of `bindds` for each individual database when binding that database's `DataSource` into the namespace.

```
bindds /test/empDS -url jdbc:oracle:thin:@empHost:5521:ORCL
      -dstype jta -dblink 2pcToEmp.oracle.com
```

Note: In a two-phase commit scenario, the `DataSource` object is bound, with respect to the two-phase commit engine. If you change the two-phase commit engine, you must update all database links, and rebind all concerned `DataSource` and `UserTransaction` objects.

4. Create the user on the two-phase commit engine that facilitates the two-phase commit. This user will open sessions to each resource involved in the transaction and complete the transaction. To do this, the user must be created on each database and granted `CONNECT`, `RESOURCE`, and `CREATE SESSION` privileges. If the user that completes the transaction is different from the user

that starts the transaction, you also need to grant the `FORCE ANY TRANSACTION` privilege. These privileges must be granted on all databases included in the transaction.

The decision on whether the `FORCE ANY TRANSACTION` privilege is needed is determined by whether you bound a username and password with the `UserTransaction` object.

- If you do not bind a username and password with the `UserTransaction`, the user that retrieved the `UserTransaction` will be the same user that is used to perform the commit or rollback for the two-phase commit on all involved databases.
- If you bind a username and password with the `UserTransaction`, then this is the username that the two-phase commit will be committed or rolled back with on all involved databases. The transaction will be started by the user that retrieves the `UserTransaction` object.

Both types of users must be created, so that it can open a session from the two-phase commit engine to each of the involved databases. Secondly, it must be granted the `CONNECT`, `RESOURCE`, `CREATE SESSION` privileges to be able to connect to each of these databases. For example, if the user that is needed for completing the transaction is `SCOTT`, you would do the following on the two-phase commit engine and each database involved in the transaction:

```
CONNECT SYSTEM/MANAGER;  
CREATE USER SCOTT IDENTIFIED BY SCOTT;  
GRANT CONNECT, RESOURCE, CREATE SESSION TO SCOTT;
```

Lastly, if you bound a username and password with the `UserTransaction` object, it will be using a different username to finalize the transaction than the username used to start the transaction. For this to be allowed, you must grant the `FORCE ANY TRANSACTION` privileges on each database involved in the transaction in order for two separate users to start and stop the transaction.

The advantage of binding a username with the `UserTransaction` is that it is treated as a global user is always committing all transactions started with this `UserTransaction` object. Thus, if you have more than one JTA transactions, you will only have to create one user and grant privileges to that user on all involved databases.

For example, if `SCOTT` is the username bound with the `UserTransaction` object, you would need to do the following in addition to the previous grant:

```
GRANT FORCE ANY TRANSACTION TO SCOTT;
```

5. Bind a `UserTransaction` into the namespace. You must provide the two-phase commit engine's fully-qualified database address. At this point, you should decide (based on the discussion in step 3) on whether to bind it with a username and password. The following assumes a global username is bound with the `UserTransaction`.

```
bindut /test/myUT -url sess_iiop://dbsun.mycompany.com:2481:ORCL
      -user SCOTT -password TIGER
```

Example 17-4 Two-Phase Commit Example

The following example shows a server object that performs an in-session activation to retrieve both the `UserTransaction` and `DataSource` objects that have been bound locally. The `UserTransaction` was bound with the two-phase commit engine's URL, username, and password. The `DataSource` objects were all bound with the proper database links.

```
//with the environment set, create the initial context.
InitialContext ic = new InitialContext ();
UserTransaction ut = (UserTransaction)ic.lookup ("/test/myUT");

//With the same username and password for the 2pc engine,
// lookup the local datasource and a remote database.
DataSource localDS = (DataSource)ic.lookup ("/test/localDS");

//remote lookup requires environment setup
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put (Context.SECURITY_PRINCIPAL, user);
env.put (Context.SECURITY_CREDENTIALS, password);
env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
env.put (jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
Context ic = new InitialContext (env);

//retrieve the DataSource for the remote database
DataSource remoteDS = (DataSource)ic.lookup ("jdbc_access://test/NewYorkDS");

//retrieve connections to both local and remote databases
Connection localConn = localDS.getConnection ();
Connection remoteConn = remoteDS.getConnection ();
...
//close the connections
localConn.close();
remoteConn.close();
```

```
//end the transaction
ut.commit();
```

Creating DataSource Objects Dynamically

If you want to bind only a single `DataSource` object in the namespace to be used for multiple database resources, you must do the following:

1. Bind the `DataSource` without specifying the URL, host, port, SID, or driver type. Thus, you execute the `bindds` tool with only the `-dstype jta` option, as follows:

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password
TIGER
& bindds /test/empDS -dstype jta
```

2. Retrieve the `DataSource` in your code. When you perform the lookup, you must cast the returned object to `OracleJTADDataSource` instead of `DataSource`. The Oracle-specific version of the `DataSource` class contains methods to set the `DataSource` properties.
3. Set the following properties:
 - Set the URL with the `OracleJTADDataSource.setURL` method
 - Fully-qualified database link if using two-phase commit engine with the `OracleJTADDataSource.setDBLink` method
4. Retrieve the connection through the `OracleJTADDataSource.getConnection` method as indicated in the other examples.

Example 17–5 Retrieving Generic DataSource

The following example retrieves a generically bound `DataSource` from the namespace using in-session lookup and initializes all relevant fields.

```
//retrieve an in-session generic DataSource object
OracleJTADDataSource ds =
    (OracleJTADDataSource)ic.lookup ("java:comp/env/test/empDS");

//set all relevant properties for my database
//URL is for a local database so use the KPRB URL
ds.setURL ("jdbc:oracle:kprb:");
//Used in two-phase commit, so provide the fully qualified database link that
```

```
//was created from the two-phase commit engine to this database
ds.setDBLink("localDB.oracle.com");

//Finally, retrieve a connection to the local database using the DataSource
Connection conn = ds.getConnection ();
```

JDBC Restrictions

If you are using JDBC calls in your server object to update a database, and you have an active transaction context, you should *not* also use JDBC to perform transaction services, by calling methods on the JDBC connection. Do *not* code JDBC transaction management methods. For example:

```
Connection conn = ...
...
conn.commit(); // DO NOT DO THIS!!
```

Doing so will cause a `SQLException` to be thrown. Instead, you must commit using the `UserTransaction` object retrieved to handle the global transaction. When you commit using the JDBC connection, you are instructing a local transaction to commit, not the global transaction. When the connection is involved in a global transaction, trying to commit a local transaction within the global transaction causes an error to occur.

In the same manner, you must also avoid doing direct SQL commits or rollbacks through JDBC. Code the object to either handle transactions directly using the `UserTransaction` interface.

Within a global transaction, you cannot execute a local transaction. If you try, the following error will be thrown:

```
ORA-2089 "COMMIT is not allowed in a subordinate session."
```

Some SQL commands implicitly execute a local transaction. All SQL DDL statements, such as "CREATE TABLE", implicitly starts and commits a local transaction under the covers. If you are involved in a global transaction that has enlisted the database that the DDL statement is executing against, the global transaction will fail.

Advanced Topics

This chapter describes the following advanced JDBC topics:

- [JDBC and Globalization Support](#)
- [JDBC Client-Side Security Features](#)
- [JDBC in Applets](#)
- [JDBC in the Server: the Server-Side Internal Driver](#)

JDBC and Globalization Support

After a brief overview, this section covers the following topics:

- [How JDBC Drivers Perform Globalization Support Conversions](#)
- [Globalization Support and Object Types](#)
- [CHAR and VARCHAR2 Data Size Restrictions with the Thin Driver](#)

Oracle's JDBC drivers support Globalization Support (formerly NLS). Globalization Support allows 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.

For more information on Globalization Support, Globalization Support environment variables, and the character sets that Oracle supports, see "[Oracle Character Datatypes Support](#)" on page 5-28 and the *Oracle9i Globalization Support Guide*. See the *Oracle9i Reference* for more information on the database character set and how it is created.

Here are a few examples of commonly used Java methods for JDBC that rely heavily on national language character set conversion:

- The `java.sql.ResultSet` methods `getString()` and `getUnicodeStream()` return values from the database as Java strings and as a stream of Unicode characters, respectively.
- The `oracle.sql.CLOB` method `getCharacterStream()` returns the contents of a CLOB as a Unicode stream.
- The `oracle.sql.CHAR` methods `getString()`, `toString()`, and `getStringWithReplacement()` convert the following data to strings:
 - `getString()`: This converts the sequence of characters represented by the CHAR object to a string and returns a Java String object.
 - `toString()`: This is identical to `getString()`, but if the character set is not recognized, then `toString()` returns a hexadecimal representation of the CHAR data.
 - `getStringWithReplacement()`: This is identical to `getString()`, except characters that have no Unicode representation in the character set of this CHAR object are replaced by a default replacement character.

How JDBC Drivers Perform Globalization Support Conversions

The techniques that the Oracle JDBC drivers use to perform character set conversion for Java applications depend on the character set the database uses. The simplest case is where the database uses the `US7ASCII` or `WE8ISO8859P1` character set. In this case, the driver converts the data directly from the database character set to `UCS-2`, which is used in Java applications, and vice versa.

If you are working with databases that employ a non-`US7ASCII` or non-`WE8ISO8859P1` character set (for example, Japanese or Korean), then the driver converts the data first to `UTF-8` (this step does not apply to the server-side internal driver), then to `UCS-2`. For example, the driver always converts `CHAR` and `VARCHAR2` data in a non-`US7ASCII`, non-`WE8ISO8859P1` character set. It does not convert `RAW` data.

Note: The JDBC drivers perform all character set conversions transparently. No user intervention is necessary for the conversions to occur.

JDBC OCI Driver and Globalization Support

For the JDBC OCI driver, the client character set is in the `NLS_LANG` environment variable, which is set at client-installation time. The language and territory settings, by default, are set to the Java VM locale settings.

Note that there are also server-side settings for these parameters, determined during database creation. So, when performing character set conversion, the JDBC OCI driver considers the following:

- database character set and language
- client character set and language
- Java applications character set: `UCS-2`

The JDBC OCI driver transfers the data from the server to the client in the character set of the database. Depending on the value of the `NLS_LANG` environment variable, the driver handles character set conversions in one of two ways:

- If `NLS_LANG` is not specified, or specifies the `US7ASCII` or `WE8ISO8859P1` character set, then the JDBC OCI driver uses Java to convert the character set from `US7ASCII` or `WE8ISO8859P1` directly to `UCS-2`, or the reverse.

or:

- If `NLS_LANG` specifies a non-US7ASCII or non-WE8ISO8859P1 character set, then the driver changes the value of the `NLS_LANG` parameter on the client to UTF-8. This happens automatically and does not require any user-intervention. OCI uses the `NLS_LANG` setting in converting the data from the database character set to UTF-8; the JDBC driver then converts the UTF-8 data to UCS-2.

Notes:

- The driver sets the `NLS_LANG` character set to UTF-8 to minimize the number of conversions it performs in Java. It performs the conversion from database character set to UTF-8 in C.
 - The change to UTF-8 is for the JDBC application process only.
 - For more information on the `NLS_LANG` parameter, see the *Oracle9i Globalization Support Guide*.
-
-

JDBC Thin Driver and Globalization Support

If you are using the JDBC Thin driver, then there will presumably be no Oracle client installation. Globalization Support conversions must be handled differently.

Language and Territory The Thin driver obtains language and territory settings (`NLS_LANGUAGE` and `NLS_TERRITORY`) from the Java locale in the JVM user `.language` property. The date format (`NLS_DATE_FORMAT`) is set according to the territory setting.

Character Set If the database character set is US7ASCII or WE8ISO8859P1, then the data is transferred to the client without any conversion. The driver then converts the character set to UCS-2 in Java.

If the database character set is something other than US7ASCII or WE8ISO8859P1, then the server first translates the data to UTF-8 before transferring it to the client. On the client, the JDBC Thin driver converts the data to UCS-2 in Java.

Server-Side Internal Driver and Globalization Support

If your JDBC code running in the server accesses the database, then the JDBC server-side internal driver performs a character set conversion based on the database character set. The target character set of all Java programs is UCS-2.

Globalization Support and Object Types

The Oracle JDBC class files, `classes12.zip` and `classes111.zip`, provide Globalization Support for the Thin and OCI drivers. The files contain all the necessary classes to provide complete Globalization Support for all Oracle character sets for CHAR and NCHAR datatypes not retrieved or inserted as part of an Oracle object or collection type. See "[Oracle Character Datatypes Support](#)" on page 5-28 for a description of CHAR and NCHAR datatypes.

However, in the case of the CHAR and VARCHAR data portion of Oracle objects and collections, the JDBC class files provide support for only the following commonly used character sets:

- US7ASCII
- WE8DEC
- ISO-LATIN-1
- UTF-8

To provide support for all national language character sets, the Oracle JDBC driver installation includes two additional files: `nls_charset12.zip` for JDK 1.2.x and `nls_charset11.zip` for JDK 1.1.x. The OCI and Thin drivers require these files to support all Oracle characters sets for CHAR and VARCHAR data in Oracle object types and collections. To obtain this support, you must add the appropriate `nls_charset*.zip` file to your CLASSPATH.

It is important to note that the `nls_charset*.zip` files are very large, because they must support a large number of character sets. To save space, you might want to keep only the classes you need from the `nls_charset*.zip` file. If you want to do this, follow these steps:

1. Unzip the appropriate `nls_charset*.zip` file.
2. Add only the needed character set classes to the CLASSPATH.
3. Remove the unneeded character set files from your system.

The character set extension class files are named in the following format:

```
CharacterConverter<OracleCharacterSetId>.class
```

where `<OracleCharacterSetId>` is the hexadecimal representation of the Oracle character set ID that corresponds to a character set name.

Note: The preceding discussion is not relevant in using the server-side internal driver, which provides complete Globalization Support and does not require the national language character set classes.

CHAR and VARCHAR2 Data Size Restrictions with the Thin Driver

If the database character set is neither ASCII (US7ASCII) nor ISO-LATIN-1 (WE8ISO8859P1), then the Thin driver must impose size restrictions for CHAR and VARCHAR2 bind parameters that are more restrictive than normal database size limitations. This is necessary to allow for data expansion during conversion.

The Thin driver checks CHAR or VARCHAR2 bind sizes when the `setXXX()` method is called. If the data size exceeds the size restriction, then the driver throws a SQL exception (ORA-17070 "Data size bigger than max size for this type") from the `setXXX()` call. This limitation is necessary to avoid the chance of data corruption whenever a Globalization Support conversion occurs and increases the length of the data. This limitation is enforced when you are doing all the following:

- using the Thin driver
- using binds (not defines)
- using CHAR or VARCHAR2 datatypes
- connecting to a database whose character set is neither ASCII (US7ASCII) nor ISO-Latin-1 (WE8ISO8859P1)

Role of the Globalization Support Ratio

As previously discussed, when the database character set is neither US7ASCII nor WE8ISO8859P1, the Thin driver converts Java UCS-2 characters to UTF-8 encoding bytes for CHAR or VARCHAR2 binds. The UTF-8 encoding bytes are then transferred to the database, and the database converts the UTF-8 encoding bytes to the database character set encoding.

This conversion to the character set encoding might result in a size increase. The Globalization Support ratio for a database character set indicates the maximum possible expansion in converting from UTF-8 to the character set:

NLS ratio = (maximum possible value of) [(size in database character set) / (size in UTF-8)]

Size Restriction Formulas

[Table 18-1](#) shows the database size limitations for CHAR and NCHAR data, and the Thin driver size restriction formulas for CHAR and NCHAR binds. Database limits are in bytes. Formulas determine the maximum size of the UTF-8 encoding, in bytes.

Table 18-1 Maximum CHAR and NCHAR Bind Sizes, Thin Driver

Oracle Version	Datatype	Max Size Allowed by Database (bytes)	Formula for Thin Driver Max Bind Size (UTF-8 bytes)
Oracle9i	NCHAR	2000	$\min(2000, 4000/\text{NLS_ratio})$
Oracle9i	NVARCHAR2	4000	$4000/\text{NLS_ratio}$
Oracle8, Oracle8i and beyond	CHAR	2000	$\min(2000, 4000/\text{NLS_ratio})$
Oracle8, Oracle8i and beyond	VARCHAR2	4000	$4000/\text{NLS_ratio}$
Oracle7	CHAR	255	255
Oracle7	VARCHAR2	2000	$2000/\text{NLS_ratio}$

The formulas guarantee that after the data is converted from UTF-8 to the database character set, the size will not exceed the database maximum size.

The number of UCS-2 characters that can be supported is determined by the number of bytes per character in the data. All ASCII characters are one byte long in UTF-8 encoding. Other character types can be two or three bytes long.

Globalization Support Ratios and Calculated Size Restrictions for Common Character Sets

[Table 18-2](#) lists the Globalization Support ratios of some common server character sets, then shows the Thin driver maximum bind sizes for CHAR and VARCHAR2 data for each character set, as determined by using the Globalization Support ratio in the appropriate formula.

Again, maximum bind sizes are for UTF-8 encoding, in bytes.

Table 18–2 Globalization Support Ratio and Size Limits, Oracle8, Common Character Sets

Server Character Set	Globalization Support Ratio	Thin Driver Max VARCHAR2 Bind Size (UTF-8 bytes)	Thin Driver Max CHAR Bind Size (UTF-8 bytes)
WE8DEC	1	4000	2000
JA16SJIS	2	2000	2000
ISO 8859–1 through 10	3	1333	1333

JDBC Client-Side Security Features

This section discusses support in the Oracle JDBC OCI and 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.

JDBC Support for Oracle Advanced Security

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

Included in your Oracle JDBC `classes111.zip` or `classes12.zip` file are a JAR file containing classes that incorporate features of Oracle Advance Security, and a JAR file containing classes whose function is to interface between the JDBC classes and the Advanced Security classes for use with the JDBC Thin driver.

OCI Driver Support for Oracle Advanced Security

If you are using one of the JDBC OCI drivers, which presumes you are running from a thick-client machine with an Oracle client installation, then support for Oracle Advanced Security and incorporated third-party features is, for the most part, no different from any Oracle thick-client situation. Your use of Advanced Security features is determined by related settings in the `SQLNET.ORA` file on the client machine, as discussed in the *Oracle Advanced Security Administrator's Guide*. Refer to that manual for information.

Important: The one key exception to the preceding, with respect to Java, is that SSL—Sun Microsystem’s standard Secure Socket Layer 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.

Thin Driver Support for Oracle Advanced Security

Because the Thin driver was designed to be downloadable with applets, one obviously cannot assume that there is an Oracle client installation and a `SQLNET.ORA` file where the Thin driver is used. This necessitated the design of a new, 100% Java approach to Oracle Advanced Security support.

Java classes that implement Oracle Advanced Security are included in your JDBC `classes12.zip` or `classes111.zip` file. Security parameters for encryption and integrity, normally set in `SQLNET.ORA`, are set in a Java properties file instead.

For information about parameter settings, see "[Thin Driver Support for Encryption and Integrity](#)" on page 18-13.

JDBC 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, as discussed in "[Open a Connection to a Database](#)" on page 3-3.

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 SecurID—are not supported by the Oracle JDBC Thin driver. For the Oracle JDBC OCI driver, support is the same as in any thick-client situation—refer to the *Oracle Advanced Security Administrator’s Guide*.

JDBC 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 an OCI driver in a thick-client setting, set parameters as you would in any Oracle client situation. When using the Thin driver, set parameters through a Java properties file.

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 18–3](#) shows how these possible settings on the client-side and server-side combine to either enable or disable the feature.

Table 18–3 *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

This table 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.

The general settings are further discussed in the *Oracle Advanced Security Administrator's Guide*. How to set them for a JDBC application is described in the following subsections.

Note: The term "checksum" still appears in integrity parameter names, as you will see in the following subsections, but is no longer used otherwise. For all intents and purposes, "checksum" and "integrity" are synonymous.

OCI Driver Support for Encryption and Integrity

If you are using one of the Oracle JDBC OCI drivers, which presumes a thick-client setting with an Oracle client installation, 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 machine.

To summarize, the client parameters are shown in [Table 18-4](#):

Table 18-4 OCI Driver Client Parameters for Encryption and Integrity

Parameter Description	Parameter Name	Possible Settings
Client encryption level	<code>SQLNET.ENCRYPTION_CLIENT</code>	REJECTED ACCEPTED REQUESTED REQUIRED
Client encryption selected list	<code>SQLNET.ENCRYPTION_TYPES_CLIENT</code>	RC4_40 RC4_56 DES DES40 (see note below)
Client integrity level	<code>SQLNET.CRYPTO_CHECKSUM_CLIENT</code>	REJECTED ACCEPTED REQUESTED REQUIRED
Client integrity selected list	<code>SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT</code>	MD5

Note: For the Oracle Advanced Security domestic edition only, a setting of `RC4_128` is also possible.

These settings, and corresponding settings in the server, are further discussed in Appendix A of the *Oracle Advanced Security Administrator's Guide*.

Thin Driver Support for Encryption and Integrity

Thin driver support for data encryption and integrity parameter settings parallels the thick-client support discussed in the preceding section. Corresponding parameters exist under the `oracle.net` package and can be set through a Java properties object that you would then use in opening your database connection.

If you replace "SQLNET" in the parameter names in [Table 18-4](#) with "oracle.net", you will get the parameter names supported by the Thin driver (but note that in Java, the parameter names are all-lowercase).

[Table 18-5](#) lists the parameter information for the Thin driver. See the next section for examples of how to set these parameters in Java.

Table 18-5 Thin Driver Client Parameters for Encryption and Integrity

Parameter Name	Parameter Type	Parameter Class	Possible Settings
<code>oracle.net.encryption_client</code>	string	static	REJECTED ACCEPTED REQUESTED REQUIRED
<code>oracle.net.encryption_types_client</code>	string	static	RC4_40 RC4_56 DES40C DES56C
<code>oracle.net.crypto_checksum_client</code>	string	static	REJECTED ACCEPTED REQUESTED REQUIRED
<code>oracle.net.crypto_checksum_types_client</code>	string	static	MD5

Notes:

- Because Oracle Advanced Security support for the Thin driver is incorporated directly into the JDBC classes ZIP 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 "C" in DES40C and DES56C refers to CBC (cipher block chaining) mode.
-
-

Setting Encryption and Integrity Parameters in Java

Use a Java properties object (`java.util.Properties`) to set the data encryption and integrity parameters supported by the Oracle JDBC Thin driver.

The following example instantiates a Java properties object, uses it to set each of the parameters in [Table 18-5](#), and then uses the properties object in opening a connection to the database:

```
...
Properties prop = new Properties();
prop.put("oracle.net.encrypted_client", "REQUIRED");
prop.put("oracle.net.encrypted_types_client", "( DES40 )");
prop.put("oracle.net.crypto_checksum_client", "REQUESTED");
prop.put("oracle.net.crypto_checksum_types_client", "( MD5 )");
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:thin:@localhost:1521:main", prop);
...
```

The parentheses around the parameter values in the `encrypted_types_client` and `crypto_checksum_types_client` settings allow for lists of values. Currently, the 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.

Complete example Following is a complete example of a class that sets data encryption and integrity parameters before connecting to a database to perform a query.

Note that in this 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.

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

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

```
{  
  
    // Register the Oracle JDBC driver  
    System.out.println("Registering the driver...");  
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());  
  
    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", "( DES40 )");  
        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.  
    Connection conn = DriverManager.getConnection  
        ("jdbc:oracle:thin:@dlsun608.us.oracle.com:1521:main", props);  
  
    // 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();  
}  
}
```

JDBC in Applets

This section describes some of the basics of working with Oracle JDBC applets, which must use the JDBC Thin driver so that an Oracle installation is not required on the client. The Thin driver connects to the database with TCP/IP protocol.

Aside from having to use the Thin driver, and being mindful of applet connection and security issues, there is essentially no difference between coding a JDBC applet and a JDBC application. There is also no difference between coding for a JDK 1.2.x browser or a JDK 1.1.x browser, other than general JDK 1.1.x to 1.2.x migration issues discussed in "[Migration from JDK 1.1.x to JDK 1.2.x](#)" on page 4-5.

This section describes what you must do for the applet to connect to a database, including how to use the Oracle8 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](#)

For general information about connecting to the database, see "[Open a Connection to a Database](#)" on page 3-3.

To see a sample applet, refer to "[Sample Applet](#)" on page 20-120.

Note: Oracle JDBC no longer supports JDK 1.0.x versions. This also applies to applets running in browsers that incorporate JDK 1.0.x versions. The user must upgrade to a browser with an environment of JDK 1.1.x or higher.

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";
conn = DriverManager.getConnection(connString, "scott", "tiger");
```

using TNS keyword-value syntax:

```
String connString = "jdbc:oracle:thin:@(description=(address_list=
    (address=(protocol=tcp)(port=1521)(host=prodHost)))
    (connect_data=(sid=ORCL)))";
conn = DriverManager.getConnection(connString, "scott", "tiger");
```

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 an Oracle database server both require many resources; you seldom find both servers running on the same machine. 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 in which you can work around the security restriction:

- You can connect to the database by using the Oracle8 Connection Manager.

or:

- You can use a signed applet to connect to the database directly.

These options are discussed in the next section, "[Connecting to a Database on a Different Host Than the Web Server](#)".

Connecting to a Database on a Different Host Than the Web Server

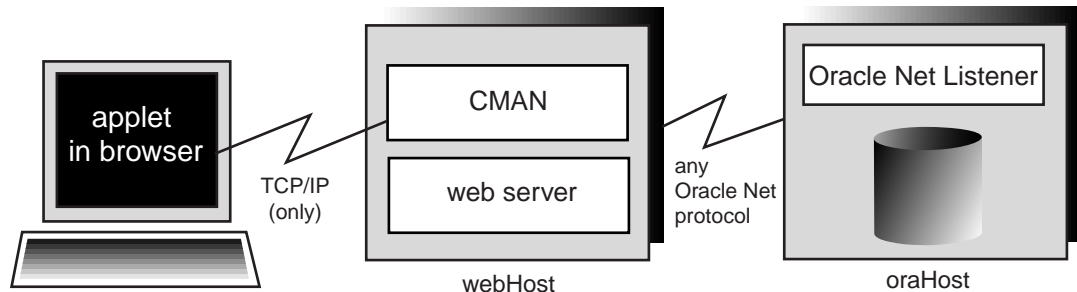
If you are connecting to a database on a host other than the one on which the Web server is running, then you must overcome applet security restrictions. You can do this by using either the Oracle8 Connection Manager or signed applets.

Using the Oracle8 Connection Manager

The Oracle8 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 18-1 illustrates the relationship between the applet, the Oracle8 Connection Manager, and the database.

Figure 18-1 Applet, Connection Manager, and Database Relationship



Using the Oracle8 Connection Manager requires two steps:

- Install and run the Connection Manager.
- Write the connection string that targets the Connection Manager.

There is also discussion of how to connect using multiple connection managers.

Installing and Running the Oracle8 Connection Manager You must install the Connection Manager, available on the Oracle8 distribution media, onto the Web server host. You can find the installation instructions in the *Oracle Net Services Administrator's Guide*.

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 this port number in your connect string for JDBC.

```
cmn = (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL=TCP)
        (HOST=<web-server-host>)
        (PORT=1610)))

cmn_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)
    )
```

Note that the Java Oracle Net version inside the JDBC Thin driver does not have authentication service support. This means that the `AUTHENTICATION_LEVEL` configuration parameter in the `CMAN.ORA` file must be set to 0.

After you create the file, start the Oracle8 Connection Manager at the operating system prompt with this command:

```
cmctl start
```

To use your applet, you must now write the connect string for it.

Writing the Connect String that Targets the Oracle8 Connection Manager This section describes how to write the connect string in your applet so that the applet connects to the Connection Manager, and the Connection Manager connects with the database. In the connect string, 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 18-1](#). 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 connect string in TNS keyword-value format:

```
Connection conn =
    DriverManager.getConnection ("jdbc:oracle:thin:" +
        "@(description=(address_list=" +
        "(address=(protocol=tcp)(host=webHost)(port=1610))" +
        "(address=(protocol=tcp)(host=oraHost)(port=1521)))" +
        "(source_route=yes)" +
        "(connect_data=(sid=orcl)))", "scott", "tiger");
```

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.

Notice that you can also write the same connect string in this format:

```
String connString =
    "jdbc:oracle:thin:@(description=(address_list=
        (address=(protocol=tcp)(port=1610)(host=webHost))
        (address=(protocol=tcp)(port=1521)(host=oraHost)))
        (connect_data=(sid=orcl))
        (source_route=yes))";
Connection conn = DriverManager.getConnection(connString, "scott", "tiger");
```

When your applet uses a connect string such as the one above, it will behave exactly as if it were connected directly to the database on the host `oraHost`.

For more information on the parameters that you specify in the connect string, see the *Oracle Net Services Administrator's Guide*.

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. See the *Oracle Net Services Administrator's Guide* for more information about `source_route` addressing.

Using Signed Applets

In either a JDK 1.2.x-based browser or a JDK 1.1.x-based 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, see Sun Microsystem's *Signed Applet Example* at:

<http://java.sun.com/security/signExample/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");
connection = DriverManager.getConnection
("jdbc:oracle:thin:scott/tiger@dlsun511:1721:orcl");
```

3. You must obtain an object-signing certificate. See Netscape's *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, see Netscape's *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 under JDK 1.2.x and 1.1.x, 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.

Firewalls are rule-based. They have a list of rules that define which clients can connect, and which cannot. Firewalls compare the client's hostname with the rules, and based on this comparison, either grant the client access, or not. If the hostname lookup fails, 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; a more detailed discussion of these firewalls is beyond the scope of this manual.

An unsigned applet can access only the same host from which it was 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, described in the following sections:

- [Configuring a Firewall for Applets that use the JDBC Thin Driver](#)
- [Writing a Connect String 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—that is, they cannot get the hostname or environment variables locally—because of security limitations. 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 two things to the firewall's list of rules:

- Add the IP address (not the hostname) of the host on which the JDBC applet is running.
- Ensure that the hostname "`__jdbc__`" never appears in the firewall's 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 Oracle's JDBC Thin driver will be able to go through your firewall.

By not including the Thin driver's hostname, the firewall is forced to do an IP address lookup and base its access decision on the IP address, instead of the hostname.

Writing a Connect String to Connect through a Firewall

To write a connect string that allows 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 through a firewall on host `fireWallHost`, listening on port 1610, then use the following connect string:

```
Connection conn =
    DriverManager.getConnection ("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=(sid=orcl)))", "scott", "tiger");
```

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

```
String connString =
    "jdbc:oracle:thin:@ixta.us.oracle.com:1521:orcl";
conn = DriverManager.getConnection (connString, "scott",
    "tiger");
```

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.

Notice that you can also write the preceding connect string in this 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=(sid=orcl))
    (source_route=yes))";
Connection conn = DriverManager.getConnection(connString, "scott", "tiger");
```

When your applet uses a connect string similar to the one above, 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 the `address_list`, the firewall address must precede the database server address.

For more information on the parameters used in the above example, see the *Oracle Net Services Administrator's Guide*. For more information on how to configure a firewall, please see your firewall's documentation or contact your firewall vendor.

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 (these will be contained in either `classes12.zip`, if you are targeting a browser that incorporates a JDK 1.2.x version, or `classes111.zip`, for a browser incorporating a JDK 1.1.x version).

Follow these steps:

1. Move the JDBC driver classes file `classes12.zip` (or `classes111.zip`) to an empty directory.

If your applet will connect to a database with a non-US7ASCII and non-WE8ISO8859P1 character set, then also move the `nls_charset12.zip` or `nls_charset11.zip` file to the same directory.
2. Unzip the JDBC driver classes ZIP file (and national language character set ZIP file, if applicable).
3. Add your applet classes files to the directory, and any other files the applet might require.
4. Zip the applet classes and driver classes together into a single ZIP or JAR file. The single zip file should contain the following:
 - class files from `classes12.zip` or `classes111.zip` (and required class files from `nls_charset12.zip` or `nls_charset11.zip` if the applet requires Globalization Support)
 - your applet classes

Additionally, if you are using `DatabaseMetaData` entry points in your applet, include the `oracle/jdbc/driver/OracleDatabaseMetaData.class` file. Note that this file is very large and might have a negative impact on performance. If you do not use `DatabaseMetaData` methods, omit this file.

5. 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>
```

You can find a description of the `APPLET`, `CODE`, `ARCHIVE`, `CODEBASE`, `WIDTH`, and `HEIGHT` parameters in the next section.

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 these parameters: `CODE`, `ARCHIVE`, `CODEBASE`, `WIDTH`, and `HEIGHT` to specify the name of the applet and its location, and the height and width of the applet display area. These parameters are described in the following sections.

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` parameters 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 applet's compiled Applet subclass—specify the file name with the `CODE` parameter. Any path 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 Applet's compiled applet subclass:

```
<APPLET CODE="JdbcApplet" WIDTH=500 HEIGHT=200>
</APPLET>
```

If you use this form of the `CODE` tag, then the classes for the applet and the classes for the JDBC Thin driver must be in the same directory as the HTML page.

Notice that in the `CODE` specification, you do not include the file name extension `".class"`.

CODEBASE

The `CODEBASE` parameter is optional and specifies the base URL of the applet; that is, the name of the directory that contains the applet's code. If it is not specified,

then the document's URL 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 entry `CODEBASE="."` indicates that the applet resides in the current directory (`my_Dir`). If the value of `codebase` was set to `Applet_Samples`, for example:

```
CODEBASE="Applet_Samples"
```

This would indicate that the applet resides in the `my_Dir/Applet_Samples` directory.

ARCHIVE

The `ARCHIVE` parameter is optional and specifies the name of the archive file (either a `.zip` or `.jar` file), if applicable, that contains the applet classes and resources the applet needs. Oracle recommends using a `.zip` file or `.jar` file, which saves many extra roundtrips to the server.

The `.zip` (or `.jar`) 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` parameter.

JDBC in the Server: the Server-Side Internal Driver

This section covers the following topics:

- [Connecting to the Database with the Server-Side Internal Driver](#)
- [Exception-Handling Extensions for the Server-Side Internal Driver](#)
- [Session and Transaction Context for the Server-Side Internal Driver](#)
- [Testing JDBC on the Server](#)
- [Server-Side Character Set Conversion of oracle.sql.CHAR Data](#)

Any Java program, Enterprise JavaBean (EJB), or Java stored procedure that runs inside the target database must use the server-side internal driver to access the local SQL engine.

This 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 invoked.

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; there is no network. This enhances the performance of your JDBC programs and is much faster than executing a remote Oracle Net call to access the SQL engine.

The server-side internal driver supports the same features, 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, you can easily move it into the database server for better performance, without having to modify the application-specific calls.

For general information about the Oracle Java platform server-side configuration or functionality, see the *Oracle9i Java Developer's Guide*.

Connecting to the Database with the Server-Side Internal Driver

As described in the preceding section, the server-side internal driver runs within a default session. You are already "connected". There are two methods you can use to access the default connection:

- Use the static `DriverManager.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, although there is no harm in doing so. This is true whether you are using `getConnection()` or `defaultConnection()` to make the connection.

Connecting with the `OracleDriver` Class `defaultConnection()` Method

The `oracle.jdbc.OracleDriver` class `defaultConnection()` method is an Oracle extension and always returns the same connection object. Even if you invoke this method multiple times, assigning the resulting connection object to different variable names, just a single connection object is reused.

You do not need to include a connect string in the `defaultConnection()` call. For example:

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

class JBCCConnection
{
    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 typically not be closed.

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

- All connection instances obtained through the `defaultConnection()` method, which actually all reference the same connection object, will be closed and unavailable for further use, with state and resource cleanup as appropriate. Executing `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 `DriverManager.getConnection()` Method

To connect to the internal server connection from code that is running within the target server, you can use the static `DriverManager.getConnection()` method with either of the following connect strings:

```
DriverManager.getConnection("jdbc:oracle:kprb:");
```

or:

```
DriverManager.getConnection("jdbc:default:connection:");
```

Any user name or password you include in the URL string is ignored in connecting to the server default connection.

The `DriverManager.getConnection()` method returns a new Java `Connection` object every time you call it. Note that although the method is not creating a new physical connection (only a single implicit connection is used), it is returning a new object.

The fact that `DriverManager.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 for the Server-Side Internal Driver

The server-side internal driver, in addition to having standard exception-handling capabilities such as `getMessage()`, `getErrorCode()`, and `getSQLState()` (as described in "[Processing SQL Exceptions](#)" on page 3-33), 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()` (standard method)

This method returns the next exception in the chain (or `null` if no further exceptions). You can start with the first exception you receive and work through the chain.

- `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.

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

This method returns a Java `Object[]` array containing the parameters included with this error.

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 for the Server-Side Internal Driver

The server-side driver operates within a default session and default transaction context. The default session is the session in which the JVM was invoked. 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();
```

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.

For example, consider the test program `JdbcCheckup.java` described in ["Testing JDBC and the Database Connection: JdbcCheckup"](#) on page 2-8. If you want to run this program on the server and connect with the `DriverManager.getConnection()` method, then open the file in your favorite text editor and change the driver name in the connection string from `"oci"` to `"kprb"`. For example:

```
Connection conn = DriverManager.getConnection  
    ("jdbc:oracle:kprb:@ " + database, user, password);
```

The advantage of using this method is that you must change only a short string in your original program. The disadvantage is that you still must provide the user, password, and database information, even though the driver will discard it. In addition, if you issue the `getConnection()` method again, the driver will create another new (and unnecessary) connection object.

However, if you connect with `defaultConnection()`, the preferred method of connecting to the database from the server-side internal driver, you do not have to

enter any user, password, or database information. You can delete these statements from your program.

For the connection statement, use:

```
Connection conn = new oracle.jdbc.OracleDriver().defaultConnection();
```

The following example is a rewrite of the `JdbcCheckup.java` program which uses the `defaultConnection()` connection statement. The connection statement is printed in bold. The unnecessary user, password, and database information statements, along with the utility function to read from standard input, have been deleted.

```
/*
 * 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 package to use JDBC
import java.sql.*;
// 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
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        Connection conn =
            new oracle.jdbc.OracleDriver ().defaultConnection ();

        // 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));
        System.out.println ("Your JDBC installation is correct.");
    }
}
```

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 compiled automatically in the server.

In either case, use the Oracle `loadjava` client-side utility to load your files. You can either specify source file names on the command line (note that the command line understands wildcards), or put the files into a JAR file and specify the JAR file name on the command line. The `loadjava` utility is discussed in detail in the *Oracle9i Java Developer's Guide*.

The `loadjava` script, which runs the actual utility, is in the `bin` subdirectory under your `[Oracle Home]` directory. This directory should already be in your path once Oracle has been installed.

Note: The `loadjava` utility does support compressed files.

Loading Class Files into the Server

Consider a case where you have three class files in your application: `Foo1.class`, `Foo2.class`, and `Foo3.class`. The following three examples demonstrate: 1) specifying the individual class file names; 2) specifying the class file names using a wildcard; and 3) specifying a JAR file that contains the class files.

Each class is written into its own class schema object in the server.

These three examples use the default OCI driver in loading the files:

```
loadjava -user scott/tiger Foo1.class Foo2.class Foo3.class
```

or:

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

or:

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

Or use the following command to load with the Thin driver (specifying the `-thin` option and an appropriate URL):

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

(Whether to use an OCI driver or the Thin driver to load classes depends on your particular environment and which performs better for you.)

Note: Because the server-side embedded JVM uses JDK 1.2.x, it is advisable to compile classes under JDK 1.2.x if they will be loaded into the server. This will catch incompatibilities during compilation, instead of at runtime (for example, JDK 1.1.x artifacts such as leftover use of the `oracle.jdbc2` package).

Loading Source Files into the Server

If you enable the `loadjava -resolve` option in 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 OCI driver:

```
loadjava -user scott/tiger -resolve Foo.java
```

Or use the following command to load with the Thin driver (specifying the `-thin` option and an appropriate URL):

```
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 for the client-side drivers, which perform character set conversions for `oracle.sql.CHAR` in Java, and offers better performance. For more information on the `oracle.sql.CHAR` class, see "[Class `oracle.sql.CHAR`](#)" on page 5-29.

Coding Tips and Troubleshooting

This chapter describes how to optimize and troubleshoot a JDBC application or applet, including the following topics:

- [JDBC and Multithreading](#)
- [Performance Optimization](#)
- [Common Problems](#)
- [Basic Debugging Procedures](#)
- [Transaction Isolation Levels and Access Modes](#)

JDBC and Multithreading

The Oracle JDBC drivers provide full support for programs that use Java multithreading. The following example creates a specified number of threads and lets you determine whether or not the threads will share a connection. If you choose to share the connection, then the same JDBC connection object will be used by all threads (each thread will have its own statement object, however).

Because all Oracle JDBC API methods are synchronized, if two threads try to use the connection object simultaneously, then one will be forced to wait until the other one finishes its use.

The program displays each thread ID and the employee name and employee ID associated with that thread.

Execute the program by entering:

```
java JdbcMTSample [number_of_threads] [share]
```

Where *number_of_threads* is the number of threads that you want to create, and *share* specifies that you want the threads to share the connection. If you do not specify the number of threads, then the program creates 10 by default.

This example is repeated in "[Multithreading—JdbcMTSample.java](#)" on page 20-20.

```
/*
 * This sample is a multi-threaded JDBC program.
 */

import java.sql.*;
import oracle.jdbc.OracleStatement;

public class JdbcMTSample extends Thread
{
    // Default no of threads to 10
    private static int NUM_OF_THREADS = 10;

    int m_myId;

    static int c_nextId = 1;
    static Connection s_conn = null;
    static boolean share_connection = false;

    synchronized static int getNextId()
    {
        return c_nextId++;
    }
}
```

```
}

public static void main (String args [])
{
    try
    {
        /* Load the JDBC driver */
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // If NoOfThreads is specified, then read it
        if ((args.length > 2) ||
            ((args.length > 1) && !(args[1].equals("share"))))
        {
            System.out.println("Error: Invalid Syntax. ");
            System.out.println("java JdbcMTSample [NoOfThreads] [share]");
            System.exit(0);
        }

        if (args.length > 1)
        {
            share_connection = true;
            System.out.println
                ("All threads will be sharing the same connection");
        }

        // get the no of threads if given
        if (args.length > 0)
            NUM_OF_THREADS = Integer.parseInt (args[0]);

        // get a shared connection
        if (share_connection)
            s_conn = DriverManager.getConnection
                ("jdbc:oracle:" +args[1], "scott","tiger");

        // Create the threads
        Thread[] threadList = new Thread[NUM_OF_THREADS];

        // spawn threads
        for (int i = 0; i < NUM_OF_THREADS; i++)
        {
            threadList[i] = new JdbcMTSample();
            threadList[i].start();
        }

        // Start everyone at the same time
    }
}
```

```
        setGreenLight ();

        // wait for all threads to end
        for (int i = 0; i < NUM_OF_THREADS; i++)
        {
            threadList[i].join();
        }

        if (share_connection)
        {
            s_conn.close();
            s_conn = null;
        }

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

public JdbcMTSample()
{
    super();
    // Assign an Id to the thread
    m_myId = getNextId();
}

public void run()
{
    Connection conn = null;
    ResultSet    rs  = null;
    Statement    stmt = null;

    try
    {
        // Get the connection

        if (share_connection)
            stmt = s_conn.createStatement (); // Create a Statement
        else
        {
            conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                             "scott", "tiger");
        }
    }
}
```

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

    while (!getGreenLight())
        yield();

    // Execute the Query
    rs = stmt.executeQuery ("select * from EMP");

    // Loop through the results
    while (rs.next())
    {
        System.out.println("Thread " + m_myId +
            " Employee Id : " + rs.getInt(1) +
            " Name : " + rs.getString(2));
        yield(); // Yield To other threads
    }

    // Close all the resources
    rs.close();
    rs = null;

    // Close the statement
    stmt.close();
    stmt = null;

    // Close the local connection
    if ((!share_connection) && (conn != null))
    {
        conn.close();
        conn = null;
    }
    System.out.println("Thread " + m_myId + " is finished. ");
}
catch (Exception e)
{
    System.out.println("Thread " + m_myId + " got Exception: " + e);
    e.printStackTrace();
    return;
}
}
static boolean greenLight = false;
static synchronized void setGreenLight () { greenLight = true; }
synchronized boolean getGreenLight () { return greenLight; }
}
```

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](#)

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.Connection` 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: Disabling AutoCommit 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.

```
// Load the Oracle JDBC driver
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

// Connect to the database
// You can put a database hostname after the @ sign in the connection URL.
Connection conn =
    DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");
```

```
// 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 allows 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.

For more information, see ["Oracle Row Prefetching"](#) on page 12-20 and ["Fetch Size"](#) on page 11-24.

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 executes a batch implicitly once a pre-set batch value is reached, or standard update batching, where the batch is executed explicitly.

For a description of the update batching models and how to use them, see ["Update Batching"](#) on page 12-2.

Common Problems

This section describes some common problems that you might encounter while using the Oracle JDBC drivers. These problems include:

- [Space Padding 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](#)

Space Padding for CHAR Columns Defined as OUT or IN/OUT Variables

In PL/SQL, CHAR columns defined as OUT or IN/OUT variables are returned to a length of 32767 bytes, padded with spaces as needed. Note that VARCHAR2 columns do not exhibit this behavior.

To avoid this problem, use the `setMaxFieldSize()` method on the `Statement` object to set a maximum limit on the length of the data that can be returned for any column. The length of the data will be the value you specify for `setMaxFieldSize()`, padded with spaces as needed. You must select the value for `setMaxFieldSize()` carefully, because this method is statement-specific and affects the length of all CHAR, RAW, LONG, LONG RAW, and VARCHAR2 columns.

To be effective, you must invoke the `setMaxFieldSize()` method before you register your OUT variables.

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 result set or 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 objects on the server side.

Boolean Parameters in PL/SQL Stored Procedures

Due to a restriction in the OCI layer, 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
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:oci8:@<...hoststring...>", "scott", "tiger");
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.

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 3-33.

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: on UNIX: TRACE_DIRECTORY_CLIENT=/oracle/traces
on Windows NT: 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.

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.

Sample Applications

This chapter presents sample applications covering a range of both standard and Oracle-specific JDBC features, categorized as follows:

- [Basic Samples](#)
- [Samples of PL/SQL in JDBC](#)
- [Intermediate Samples](#)
- [Samples for JDBC 2.0 Types](#)
- [Samples for Oracle Type Extensions](#)
- [Samples for Custom Object Classes](#)
- [JDBC 2.0 Result Set Enhancement Samples](#)
- [Performance Enhancement Samples](#)
- [Samples for Connection Pooling and Distributed Transactions](#)
- [Sample Applet](#)
- [JDBC versus SQLJ Sample Code](#)

These samples are located in subdirectories under the following directory on the product CD:

```
[Oracle Home]/jdbc/demo/samples
```

Note: Aside from the sample applet, which requires the Thin driver, all samples in this chapter work with any JDBC driver. Do not be misled by the fact that most of the samples are located under the `oci8` directory on the product CD.

Basic Samples

This section provides elementary samples that print employee information from a table and insert employee information into the table.

- [Listing Names from the EMP Table—Employee.java](#)
- [Inserting Names into the EMP Table—InsertExample.java](#)

These samples are located in the following directory on the product CD:

```
[Oracle Home]/jdbc/demo/samples/oci8/basic-samples
```

For a step-by-step discussion of basic JDBC functionality, see ["First Steps in JDBC"](#) on page 3-2.

Listing Names from the EMP Table—Employee.java

This example retrieves and prints all the employee names from the EMP table.

Note: Do not confuse this `Employee.java` with the one used later as an example of a `ORADATA` implementation of a custom Java class.

```
/*
 * This sample shows how to list all the names from the EMP table
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

class Employee
{
    public static void main (String args [])
        throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");
```



```

// 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));

// Close the ResultSet
rset.close();

// Close the Statement
stmt.close();

// Close the connection
conn.close();
}
}

```

Inserting Names into the EMP Table—InsertExample.java

This sample uses a prepared statement to insert new employee rows into the EMP table.

```

/*
 * This sample shows how to insert data in a table.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

class InsertExample
{
    public static void main (String args [])
        throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =

```

```
        DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

// Prepare a statement to cleanup the emp table
Statement stmt = conn.createStatement ();
try
{
    stmt.execute ("delete from EMP where EMPNO = 1500");
}
catch (SQLException e)
{
    // Ignore an error here
}

try
{
    stmt.execute ("delete from EMP where EMPNO = 507");
}
catch (SQLException e)
{
    // Ignore an error here too
}

// Close the statement
stmt.close();

// 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();

// Close the connection
```

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

Samples of PL/SQL in JDBC

The following examples demonstrate the interoperability between PL/SQL and JDBC, contrasting standard SQL92 calling syntax with Oracle PL/SQL block syntax:

- [Executing Procedures in PL/SQL Blocks—PLSQL.java](#)
- [Calling PL/SQL Stored Procedures—PLSQLExample.java](#)

These samples are located in the following directory on the product CD:

```
[Oracle Home]/jdbc/demo/samples/oci8/basic-samples
```

For related discussion, see "[PL/SQL Stored Procedures](#)" on page 3-31.

Calling PL/SQL Stored Procedures—PLSQLExample.java

This sample defines a stored function and executes it using SQL92 CALL syntax in a callable statement. The function takes an employee name and salary as input and raises the salary by a set amount.

```
/*
 * This sample shows how to call a PL/SQL stored procedure using the SQL92
 * syntax. See also the other sample PLSQL.java.
 */

import java.sql.*;
import java.io.*;

class PLSQLExample
{
    public static void main (String args [])
        throws SQLException, IOException
    {
        // Load the driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

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

        // Create the stored function
```

```

stmt.execute ("create or replace function RAISESAL (name CHAR, raise NUMBER)
              return NUMBER is begin return raise + 100000; end;");

// Close the statement
stmt.close();

// Prepare to call the stored procedure RAISESAL.
// This sample uses the SQL92 syntax
CallableStatement cstmt = conn.prepareCall (" {? = call RAISESAL (?, ?)}");

// Declare that the first ? is a return value of type Int
cstmt.registerOutParameter (1, Types.INTEGER);

// We want to raise LESLIE's salary by 20,000
cstmt.setString (2, "LESLIE"); // The name argument is the second ?
cstmt.setInt (3, 20000);       // The raise argument is the third ?

// Do the raise
cstmt.execute ();

// Get the new salary back
int new_salary = cstmt.getInt (1);

System.out.println ("The new salary is: " + new_salary);

// Close the statement
cstmt.close();

// Close the connection
conn.close();
}
}

```

Executing Procedures in PL/SQL Blocks—PLSQL.java

This sample defines PL/SQL stored procedures and functions and executes them from within Oracle PL/SQL `BEGIN . . . END` blocks in callable statements. Stored procedures and functions with input, output, input-output, and return parameters are shown.

```

/*
 * This sample shows how to call PL/SQL blocks from JDBC.
 */

```

```
import java.sql.*;

class PLSQL
{
    public static void main (String args [])
        throws SQLException, ClassNotFoundException
    {
        // Load the driver
        Class.forName ("oracle.jdbc.OracleDriver");

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        // Create the stored procedures
        init (conn);

        // Cleanup the plsqttest database
        Statement stmt = conn.createStatement ();
        stmt.execute ("delete from plsqttest");

        // Close the statement
        stmt.close();

        // Call a procedure with no parameters
        {
            CallableStatement procnone = conn.prepareCall ("begin procnone; end;");
            procnone.execute ();
            dumpTestTable (conn);
            procnone.close();
        }

        // Call a procedure with an IN parameter
        {
            CallableStatement procin = conn.prepareCall ("begin procin (?); end;");
            procin.setString (1, "testing");
            procin.execute ();
            dumpTestTable (conn);
            procin.close();
        }

        // Call a procedure with an OUT parameter
        {
            CallableStatement procout = conn.prepareCall ("begin procout (?); end;");
```

```
        procout.registerOutParameter (1, Types.CHAR);
        procout.execute ();
        System.out.println ("Out argument is: " + procout.getString (1));
        procout.close();
    }

    // Call a procedure with an IN/OUT parameter
    {
        CallableStatement procinout = conn.prepareCall
            ("begin procinout (?); end;");
        procinout.registerOutParameter (1, Types.VARCHAR);
        procinout.setString (1, "testing");
        procinout.execute ();
        dumpTestTable (conn);
        System.out.println ("Out argument is: " + procinout.getString (1));
        procinout.close();
    }

    // Call a function with no parameters
    {
        CallableStatement funcnone = conn.prepareCall
            ("begin ? := funcnone; end;");
        funcnone.registerOutParameter (1, Types.CHAR);
        funcnone.execute ();
        System.out.println ("Return value is: " + funcnone.getString (1));
        funcnone.close();
    }

    // Call a function with an IN parameter
    {
        CallableStatement funcin = conn.prepareCall
            ("begin ? := funcin (?); end;");
        funcin.registerOutParameter (1, Types.CHAR);
        funcin.setString (2, "testing");
        funcin.execute ();
        System.out.println ("Return value is: " + funcin.getString (1));
        funcin.close();
    }

    // Call a function with an OUT parameter
    {
        CallableStatement funcout = conn.prepareCall
            ("begin ? := funcout (?); end;");
        funcout.registerOutParameter (1, Types.CHAR);
        funcout.registerOutParameter (2, Types.CHAR);
    }
```

```
        funcout.execute ();
        System.out.println ("Return value is: " + funcout.getString (1));
        System.out.println ("Out argument is: " + funcout.getString (2));
        funcout.close();
    }

    // Close the connection
    conn.close();
}

// Utility function to dump the contents of the PLSQLTEST table and
// clear it
static void dumpTestTable (Connection conn)
    throws SQLException
{
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("select * from plsqttest");
    while (rset.next ())
        System.out.println (rset.getString (1));
    stmt.execute ("delete from plsqttest");
    rset.close();
    stmt.close();
}

// Utility function to create the stored procedures
static void init (Connection conn)
    throws SQLException
{
    Statement stmt = conn.createStatement ();
    try { stmt.execute ("drop table plsqttest"); } catch (SQLException e) { }
    stmt.execute ("create table plsqttest (x char(20))");
    stmt.execute ("create or replace procedure procnone
        is begin insert into plsqttest values ('testing'); end;");
    stmt.execute ("create or replace procedure procin (y char)
        is begin insert into plsqttest values (y); end;");
    stmt.execute ("create or replace procedure procout (y out char)
        is begin y := 'tested'; end;");
    stmt.execute ("create or replace procedure procinout (y in out varchar)
        is begin insert into plsqttest values (y);
        y := 'tested'; end;");

    stmt.execute ("create or replace function funcnone return char
        is begin return 'tested'; end;");
    stmt.execute ("create or replace function funcin (y char) return char
        is begin return y || y; end;");
}
```



```

        stmt.execute ("create or replace function funcout (y out char) return char
                    is begin y := 'tested'; return 'returned'; end;");
        stmt.close();
    }
}

```

Accessing a PL/SQL Index-by Table from JDBC—PLSQLIndexTab.java

This code example shows special input binding and output registration methods in the IN, OUT (including function return values), and IN OUT parameter modes.

For a complete discussion of this topic, see ["Accessing PL/SQL Index-by Tables"](#) on page 16-21.

```

/*
 * This sample demonstrates how to make PL/SQL calls with index-by table
 * parameters
 */

// You need to import java.sql, oracle.sql and oracle.jdbc packages to use
import java.sql.*;
import oracle.jdbc.*;
import oracle.sql.*;

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

        String [] plSqlIndexArrayIn = {"string1","string2","string3"};
        int currentLen = plSqlIndexArrayIn.length;
        int maxLen = currentLen;
        int elementMaxLen = 20;

        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        String url = "jdbc:oracle:oci8:@";
        try {
            String url1 = System.getProperty("JDBC_URL");
            if (url1 != null)
                url = url1;
        } catch (Exception e) {
            // If there is any security exception, ignore it

```

```
        // and use the default
    }

    // Connect to the database
    Connection conn =
        DriverManager.getConnection (url, "scott", "tiger");

    // Create the procedures which use Index-by Table as IN/OUT parameters
    createProc_Func(conn);

    // Call a procedure with an IN parameter
    System.out.println ("Call a procedure with an IN parameter");
    OracleCallableStatement cs =
        (OracleCallableStatement) conn.prepareStatement ("begin proc_in (?, ?);
end;");

    // Use setPlsqlIndexTable() to set the Index-by Table parameter
    cs.setPlsqlIndexTable (1, plSqlIndexArrayIn, maxLen,
        currentLen, OracleTypes.VARCHAR, elementMaxLen);

    // Register OUT parameter
    cs.registerOutParameter (2, Types.CHAR);

    // Call the procedure
    cs.execute ();

    // Display the Status
    System.out.println ("Status = " + cs.getString (2));

    // Call a procedure with an OUT parameter
    System.out.println ("Call a procedure with an OUT parameter");
    cs = (OracleCallableStatement) conn.prepareStatement ("begin proc_out (?);
end;");

    // Use setPlsqlIndexTable() to set the Index-by Table parameter
    cs.registerIndexTableOutParameter (1, maxLen, OracleTypes.VARCHAR,
    elementMaxLen);

    // Call the procedure
    cs.execute ();

    // Display the OUT value
    Datum[] val = cs.getOraclePlsqlIndexTable (1);
    for (int i = 0; i < val.length; i++)
        System.out.println ("Value = " + val[i].stringValue());
```

```
// Call a procedure with IN/OUT parameter
System.out.println ("Call a procedure with IN/OUT parameter");

cs = (OracleCallableStatement) conn.prepareCall ("begin proc_inout (?, ?);
end;");

// Use setPlsqlIndexTable() to set the Index-by Table parameter
cs.setPlsqlIndexTable (1, plSqlIndexArrayIn, maxLen,
                       currentLen, OracleTypes.VARCHAR, elementMaxLen);

// Register OUT parameter
cs.registerIndexTableOutParameter (1, maxLen, OracleTypes.VARCHAR,
elementMaxLen);
cs.registerOutParameter (2, Types.CHAR);

// Call the procedure
cs.execute ();

// Display the Status
System.out.println ("Status = " + cs.getString (2));

// Display the OUT value
val = cs.getOraclePlsqlIndexTable (1);
for (int i = 0; i < val.length; i++)
    System.out.println ("Value = " + val[i].stringValue());

// Call the Function
System.out.println ("Call the function");

cs = (OracleCallableStatement) conn.prepareCall ("begin ? := func (?);
end;");

// Use setPlsqlIndexTable() to set the Index-by Table parameter
cs.setPlsqlIndexTable (2, plSqlIndexArrayIn, maxLen,
                       currentLen, OracleTypes.VARCHAR, elementMaxLen);

// Register OUT parameter
cs.registerIndexTableOutParameter (1, maxLen, OracleTypes.VARCHAR,
elementMaxLen);

// Call the procedure
cs.execute ();

val = cs.getOraclePlsqlIndexTable (1);
```

```
        for (int i = 0; i < val.length; i++)
            System.out.println ("Value = " + val[i].stringValue());

        // Close the Callable Statement
        cs.close();

        // Dump the contents of the demo_tab
        System.out.println ("Dump the demo_tab table");
        dumpTable(conn);

        // Clean up the schema
        cleanup(conn);

        // Close the connection
        conn.close();
    }

    private static void createProc_Func (Connection conn)
        throws SQLException
    {
        // Cleanup the schema
        cleanup(conn);

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

        // Create the Table
        stmt.execute("CREATE TABLE demo_tab (coll VARCHAR2(20))");

        // Use PL/SQL to create the Package
        String plsql1 = "CREATE OR REPLACE PACKAGE pkg AS " +
            " TYPE indexByTab IS TABLE OF VARCHAR2(20) INDEX BY" +
            " BINARY_INTEGER; " +
            "END;";

        stmt.execute(plsql1);

        // Create a procedure to use the Index-by Table as IN parameter
        String plsql2 = "CREATE OR REPLACE PROCEDURE proc_in (p1 IN pkg.indexByTab," +
            status OUT VARCHAR2) IS " +
            "BEGIN " +
            " FOR i in 1..3 LOOP " +
            "     INSERT INTO demo_tab VALUES (p1(i)); " +
            " END LOOP; " +
            " IF ((p1(1)='string1') AND (p1(2)='string2') AND
```

```

        (pl(3)='string3')) " +
        "     THEN status := 'Values passed in correctly'; " +
        " ELSE " +
        "     status := 'Values passed in are incorrect'; " +
        " END IF; " +
        "END;";

stmt.execute(plsql2);

// Create a procedure to use the Index-by Table as OUT parameter
String plsql3 = "CREATE OR REPLACE PROCEDURE proc_out (p1 OUT
    pkg.indexByTab) IS " +
    "BEGIN " +
    "  p1(1) := 'string1'; " +
    "  p1(2) := 'string2'; " +
    "  p1(3) := 'string3'; " +
    "END;";

stmt.execute(plsql3);

// Create a procedure to use the Index-by Table as both IN and OUT parameter
String plsql4 = "CREATE OR REPLACE PROCEDURE proc_inout (p1 IN OUT
    pkg.indexByTab, status OUT VARCHAR2) IS " +
    "BEGIN " +
    "  FOR i in 1..3 LOOP " +
    "    INSERT INTO demo_tab VALUES (p1(i)); " +
    "  END LOOP; " +
    "  IF ((p1(1)='string1') AND (p1(2)='string2') AND
        (p1(3)='string3')) " +
    "    THEN status := 'Values passed in correctly'; " +
    "  ELSE " +
    "    status := 'Values passed in are incorrect'; " +
    "  END IF; " +
    "  p1(1) := 'string4'; " +
    "  p1(2) := 'string5'; " +
    "  p1(3) := 'string6'; " +
    "END;";

stmt.execute(plsql4);

String plsql5 = "CREATE OR REPLACE FUNCTION func (p1 pkg.indexByTab) RETURN
    pkg.indexByTab IS " +
    "  n pkg.indexByTab; " +
    "BEGIN " +
    "  FOR i in 1..3 LOOP " +

```

```
        "    INSERT INTO demo_tab VALUES (p1(i)); " +
        " END LOOP; " +
        " IF ((p1(1)='string1') AND (p1(2)='string2') AND
        (p1(3)='string3')) THEN " +
        "    n(1) := 'p1(1) correct'; " +
        "    n(2) := 'p1(2) correct'; " +
        "    n(3) := 'p1(3) correct'; " +
        " ELSE " +
        "    n(1) := 'p1(1) wrong'; " +
        "    n(2) := 'p1(2) wrong'; " +
        "    n(3) := 'p1(3) wrong'; " +
        " END IF; " +
        " RETURN n; " +
        "END;";

stmt.execute(plsql5);

// Close the statement
stmt.close();
}

/**
 * Utility function to dump the contents of the "demo_tab" table
 */
static void dumpTable (Connection conn) throws SQLException
{
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("select * from demo_tab");
    while (rset.next ())
        System.out.println (rset.getString (1));
    rset.close();
    stmt.close();
}

/**
 * Cleanup data structures created in this example
 */
static void cleanup (Connection conn) throws SQLException
{
    Statement stmt = conn.createStatement ();

    try {
        stmt.execute ("DROP TABLE demo_tab");
    } catch (SQLException e) {}
}
```

```
try {
    stmt.execute ("DROP PROCEDURE proc_in");
} catch (SQLException e) {}

try {
    stmt.execute ("DROP PROCEDURE proc_out");
} catch (SQLException e) {}

try {
    stmt.execute ("DROP PROCEDURE proc_inout");
} catch (SQLException e) {}

try {
    stmt.execute ("DROP PROCEDURE func");
} catch (SQLException e) {}

try {
    stmt.execute ("DROP PACKAGE pck");
} catch (SQLException e) {}

stmt.close ();
}
}
```

Intermediate Samples

Samples in this section demonstrate intermediate-level JDBC functionality.

- [Streams—StreamExample.java](#)
- [Multithreading—JdbcMTSample.java](#)

These samples are located in the following directory on the product CD:

```
[Oracle Home]/jdbc/demo/samples/oci8/basic-samples
```

Streams—StreamExample.java

The JDBC drivers support the manipulation of data streams in both directions between client and server. The code sample in this section demonstrates this by connecting to a database and inserting and fetching LONG data using standard JDBC stream API.

For a complete discussion of this topic, see "[Java Streams in JDBC](#)" on page 3-19.

```
/*
 * This example shows how to stream data from the database
 */

import java.sql.*;
import java.io.*;

class StreamExample
{
    public static void main (String args [])
        throws SQLException, IOException
    {
        // Load the driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        // It's faster when you don't commit automatically
        conn.setAutoCommit (false);

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



```
// Create the example table
try
{
    stmt.execute ("drop table streamexample");
}
catch (SQLException e)
{
    // An exception would be raised if the table did not exist
    // We just ignore it
}

// Create the table
stmt.execute ("create table streamexample
              (NAME varchar2 (256), DATA long)");

// Let's insert some data into it. We'll put the source code
// for this very test in the database.
File file = new File ("StreamExample.java");
InputStream is = new FileInputStream ("StreamExample.java");
PreparedStatement pstmt =
    conn.prepareStatement ("insert into streamexample
                          (data, name) values (?, ?)");
pstmt.setAsciiStream (1, is, (int)file.length ());
pstmt.setString (2, "StreamExample");
pstmt.execute ();

// Do a query to get the row with NAME 'StreamExample'
ResultSet rset =
    stmt.executeQuery ("select DATA from streamexample where
                      NAME='StreamExample'");

// Get the first row
if (rset.next ())
{
    // Get the data as a Stream from Oracle to the client
    InputStream gif_data = rset.getAsciiStream (1);

    // Open a file to store the gif data
    FileOutputStream os = new FileOutputStream ("example.out");

    // Loop, reading from the gif stream and writing to the file
    int c;
    while ((c = gif_data.read ()) != -1)
        os.write (c);
}
```

```
        // Close the file
        os.close ();
    }

    // Close all the resources
    if (rset != null)
        rset.close();

    if (stmt != null)
        stmt.close();

    if (pstmt != null)
        pstmt.close();

    if (conn != null)
        conn.close();
    }
}
```

Multithreading—JdbcMTSample.java

The Oracle JDBC drivers provide full support for programs that use Java multithreading. The following sample program creates a specified number of threads and lets you determine whether or not the threads will share a connection. If you choose to share the connection, then the same JDBC connection object will be used by all threads (each thread will have its own statement object, however).

Because all Oracle JDBC API methods (except the `cancel()` method) are synchronized, if two threads try to use the connection object simultaneously, then one will be forced to wait until the other one finishes its use.

The program displays each thread ID and the employee name and employee ID associated with that thread.

This sample is repeated in ["JDBC and Multithreading"](#) on page 19-2.

```
/*
 * This sample is a multi-threaded JDBC program.
 */

import java.sql.*;
import oracle.jdbc.OracleStatement;

public class JdbcMTSample extends Thread
```

```
{
// Default no of threads to 10
private static int NUM_OF_THREADS = 10;

int m_myId;

static int c_nextId = 1;
static Connection s_conn = null;
static boolean share_connection = false;

synchronized static int getNextId()
{
    return c_nextId++;
}

public static void main (String args [])
{
    try
    {
        /* Load the JDBC driver */
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // If NoOfThreads is specified, then read it
        if ((args.length > 2) ||
            ((args.length > 1) && !(args[1].equals("share"))))
        {
            System.out.println("Error: Invalid Syntax. ");
            System.out.println("java JdbcMTSample [NoOfThreads] [share]");
            System.exit(0);
        }

        if (args.length > 1)
        {
            share_connection = true;
            System.out.println
                ("All threads will be sharing the same connection");
        }

        // get the no of threads if given
        if (args.length > 0)
            NUM_OF_THREADS = Integer.parseInt (args[0]);

        // get a shared connection
        if (share_connection)
            s_conn = DriverManager.getConnection
```

```
        ("jdbc:oracle:oci8:@", "scott","tiger");

    // Create the threads
    Thread[] threadList = new Thread[NUM_OF_THREADS];

    // spawn threads
    for (int i = 0; i < NUM_OF_THREADS; i++)
    {
        threadList[i] = new JdbcMTSample();
        threadList[i].start();
    }

    // Start everyone at the same time
    setGreenLight ();

    // wait for all threads to end
    for (int i = 0; i < NUM_OF_THREADS; i++)
    {
        threadList[i].join();
    }

    if (share_connection)
    {
        s_conn.close();
        s_conn = null;
    }
}
catch (Exception e)
{
    e.printStackTrace();
}

}

public JdbcMTSample()
{
    super();
    // Assign an Id to the thread
    m_myId = getNextId();
}

public void run()
{
    Connection conn = null;
```

```
ResultSet    rs    = null;
Statement    stmt = null;

try
{
    // Get the connection

    if (share_connection)
        stmt = s_conn.createStatement (); // Create a Statement
    else
    {
        conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                           "scott","tiger");
        stmt = conn.createStatement (); // Create a Statement
    }

    while (!getGreenLight())
        yield();

    // Execute the Query
    rs = stmt.executeQuery ("select * from EMP");

    // Loop through the results
    while (rs.next())
    {
        System.out.println("Thread " + m_myId +
                           " Employee Id : " + rs.getInt(1) +
                           " Name : " + rs.getString(2));
        yield(); // Yield To other threads
    }

    // Close all the resources
    rs.close();
    rs = null;

    // Close the statement
    stmt.close();
    stmt = null;

    // Close the local connection
    if ((!share_connection) && (conn != null))
    {
        conn.close();
        conn = null;
    }
}
```

```
        System.out.println("Thread " + m_myId + " is finished. ");
    }
    catch (Exception e)
    {
        System.out.println("Thread " + m_myId + " got Exception: " + e);
        e.printStackTrace();
        return;
    }
}

static boolean greenLight = false;
static synchronized void setGreenLight () { greenLight = true; }
synchronized boolean getGreenLight () { return greenLight; }

}
```

Samples for JDBC 2.0 Types

This section contains sample code for the Oracle implementations of standard JDBC 2.0 types:

- [BLOBs and CLOBs—LobExample.java](#)
- [Weakly Typed Objects—PersonObject.java](#)
- [Weakly Typed Object References—StudentRef.java](#)
- [Weakly Typed Arrays—ArrayExample.java](#)

These samples are located in the following directory on the product CD:

```
[Oracle Home]/jdbc/demo/samples/oci8/object-samples
```

BLOBs and CLOBs—LobExample.java

This sample demonstrates basic JDBC support for LOBs. It illustrates how to create a table containing LOB columns and includes utility programs to read from a LOB, write to a LOB, and dump the LOB contents. For more information on LOBs, see "[Working with BLOBs and CLOBs](#)" on page 7-3.

```
/*
 * This sample demonstrate basic LOB support.
 */

import java.sql.*;
import java.io.*;
import java.util.*;
import oracle.jdbc.driver.*;

//needed for new CLOB and BLOB classes
import oracle.sql.*;

public class LobExample
{
    public static void main (String args [])
        throws Exception
    {
        // Register the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection
```

```
URL.  
    Connection conn =  
        DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott",  
"tiger");  
  
    // Need to set auto commit off to update LOBs  
    conn.setAutoCommit (false);  
  
    // Create a Statement  
    Statement stmt = conn.createStatement ();  
  
    try  
    {  
        stmt.execute ("drop table basic_lob_table");  
    }  
    catch (SQLException e)  
    {  
        // An exception could be raised here if the table did not exist  
already.  
    }  
  
    // Create a table containing a BLOB and a CLOB  
    stmt.execute ("create table basic_lob_table (x varchar2 (30), b  
blob, c clob)");  
  
    // Populate the table  
    stmt.execute ("insert into basic_lob_table values ('one',  
'01010101010101010101010101010101', 'onetwothreefour')");  
    stmt.execute ("insert into basic_lob_table values ('two',  
'02020202020202020202020202020202', 'twothreefourfivesix')");  
  
    // commit set up  
    conn.commit();  
  
    System.out.println ("Dumping lob");  
  
    // Select the lob  
    // note that the FOR UPDATE clause is needed for updating LOBs  
    ResultSet rset = stmt.executeQuery ("select * from basic_lob_table  
for update");  
    while (rset.next ())  
    {  
        // Get the lob  
        BLOB blob = ((OracleResultSet)rset).getBLOB (2);  
        CLOB clob = ((OracleResultSet)rset).getCLOB (3);
```



```
// Print the lob contents
dumpBlob (conn, blob);
dumpClob (conn, clob);

// Change the lob contents
fillClob (conn, clob, 2000);
fillBlob (conn, blob, 4000);

}
// You could rollback the changes made by fillClob() and fillBlob()
// by issuing a rollback here
// conn.rollback();

System.out.println ("Dumping lobes again");

// No need to have FOR UPDATE clause just to do selects
rset = stmt.executeQuery ("select * from basic_lob_table");
while (rset.next ())
{
    // Get the lobes
    BLOB blob = ((OracleResultSet)rset).getBLOB (2);
    CLOB clob = ((OracleResultSet)rset).getCLOB (3);

    // Print the lobes contents
    dumpBlob (conn, blob);
    dumpClob (conn, clob);
}
// Close all resources
rset.close();
stmt.close();
conn.close();
}

// Utility function to dump Clob contents
static void dumpClob (Connection conn, CLOB clob)
    throws Exception
{
    // get character stream to retrieve clob data
    Reader instream = clob.getCharacterStream();

    // create temporary buffer for read
    char[] buffer = new char[10];

    // length of characters read
```

```
int length = 0;

// fetch data
while ((length = instream.read(buffer)) != -1)
{
    System.out.print("Read " + length + " chars: ");

    for (int i=0; i<length; i++)
        System.out.print(buffer[i]);
    System.out.println();
}

// Close input stream
instream.close();
}

// Utility function to dump Blob contents
static void dumpBlob (Connection conn, BLOB blob)
    throws Exception
{
    // Get binary output stream to retrieve blob data
    InputStream instream = blob.getBinaryStream();

    // Create temporary buffer for read
    byte[] buffer = new byte[10];

    // length of bytes read
    int length = 0;

    // Fetch data
    while ((length = instream.read(buffer)) != -1)
    {
        System.out.print("Read " + length + " bytes: ");

        for (int i=0; i<length; i++)
            System.out.print(buffer[i]+" ");
        System.out.println();
    }

    // Close input stream
    instream.close();
}

// Utility function to put data in a Clob
static void fillClob (Connection conn, CLOB clob, long length)
```

```
throws Exception
{
    Writer outstream = clob.getCharacterOutputStream();

    int i = 0;
    int chunk = 10;

    while (i < length)
    {
        outstream.write(i + "hello world", 0, chunk);

        i += chunk;
        if (length - i < chunk)
            chunk = (int) length - i;
    }
    outstream.close();
}

// Utility function to put data in a Blob
static void fillBlob (Connection conn, BLOB blob, long length)
    throws Exception
{
    OutputStream outstream = blob.getBinaryOutputStream();

    int i = 0;
    int chunk = 10;

    byte [] data = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    while (i < length)
    {
        data [0] = (byte)i;
        outstream.write(data, 0, chunk);

        i += chunk;
        if (length - i < chunk)
            chunk = (int) length - i;
    }
    outstream.close();
}
}
```

Weakly Typed Objects—PersonObject.java

This sample demonstrates the functionality of the Oracle classes `oracle.sql.STRUCT` and `oracle.sql.StructDescriptor` for weakly typed support of SQL structured objects. It defines the SQL object types `PERSON` and `ADDRESS` (an attribute of `PERSON`).

For a complete discussion of weakly typed `STRUCT` class functionality, see ["Using the Default STRUCT Class for Oracle Objects"](#) on page 8-3.

```
/*
 * This sample demonstrate basic Object support
 */

import java.sql.*;
import java.io.*;
import java.util.*;
import java.math.BigDecimal;
import oracle.sql.*;
import oracle.jdbc.*;

public class PersonObject
{
    public static void main (String args [])
        throws Exception
    {
        // Register the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You need to put your database name after the @ sign in
        // the connection URL.
        //
        // The sample retrieves an object of type "STUDENT",
        // materializes the object as an object of type ADT.
        // The Object is then modified and inserted back into the database.

        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@",
                                        "scott", "tiger");

        // It's faster when auto commit is off
        conn.setAutoCommit (false);

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

```
try
{
    stmt.execute ("drop table people");
    stmt.execute ("drop type PERSON FORCE");
    stmt.execute ("drop type ADDRESS FORCE");
}
catch (SQLException e)
{
    // the above drop and create statements will throw exceptions
    // if the types and tables did not exist before
}

stmt.execute ("create type ADDRESS as object
              (street VARCHAR (30), num NUMBER)");
stmt.execute ("create type PERSON as object
              (name VARCHAR (30), home ADDRESS)");
stmt.execute ("create table people (empno NUMBER, empid PERSON)");

stmt.execute ("insert into people values
              (101, PERSON ('Greg', ADDRESS ('Van Ness', 345)))");
stmt.execute ("insert into people values
              (102, PERSON ('John', ADDRESS ('Geary', 229)))");

ResultSet rs = stmt.executeQuery ("select * from people");
showResultSet (rs);
rs.close();

//now insert a new row

// create a new STRUCT object with a new name and address
// create the embedded object for the address
Object [] address_attributes = new Object [2];
address_attributes [0] = "Mission";
address_attributes [1] = new BigDecimal (346);

StructDescriptor addressDesc =
    StructDescriptor.createDescriptor ("ADDRESS", conn);
STRUCT address = new STRUCT (addressDesc, conn, address_attributes);

Object [] person_attributes = new Object [2];
person_attributes [0] = "Gary";
person_attributes [1] = address;

StructDescriptor personDesc =
```

```
        StructDescriptor.createDescriptor("PERSON", conn);
        STRUCT new_person = new STRUCT (personDesc, conn, person_attributes);

        PreparedStatement ps =
            conn.prepareStatement ("insert into people values (?,?)");
        ps.setInt (1, 102);
        ps.setObject (2, new_person);

        ps.execute ();
        ps.close();

        rs = stmt.executeQuery ("select * from people");
        System.out.println ();
        System.out.println (" a new row has been added to the people table");
        System.out.println ();
        showResultSet (rs);

        rs.close();
        stmt.close();
        conn.close();
    }

    public static void showResultSet (ResultSet rs)
        throws SQLException
    {
        while (rs.next ())
        {
            int empno = rs.getInt (1);
            // retrieve the STRUCT
            STRUCT person_struct = (STRUCT)rs.getObject (2);
            Object person_attrs[] = person_struct.getAttributes();

            System.out.println ("person name: " + (String) person_attrs[0]);

            STRUCT address = (STRUCT) person_attrs[1];

            System.out.println ("person address: ");

            Object address_attrs[] = address.getAttributes();

            System.out.println ("street: " + (String) address_attrs[0]);
            System.out.println ("number: " +
                ((BigDecimal) address_attrs[1]).intValue());
            System.out.println ();
        }
    }
}
```

```

    }
}

```

Weakly Typed Object References—StudentRef.java

This sample demonstrates the functionality of the Oracle class `oracle.sql.REF` for weakly typed support of SQL object references. It defines the SQL object type `STUDENT` and uses references to that object type.

For a complete discussion of weakly typed `REF` class functionality, see [Chapter 9, "Working with Oracle Object References"](#).

```

/*
 * This sample demonstrate basic Ref support
 */

import java.sql.*;
import java.io.*;
import java.util.*;
import java.math.BigDecimal;
import oracle.sql.*;
import oracle.jdbc.*;

public class StudentRef
{
    public static void main (String args [])
        throws Exception
    {
        // Register the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You need to put your database name after the @ sign in
        // the connection URL.
        //
        // The sample retrieves an object of type "person",
        // materializes the object as an object of type ADT.
        // The Object is then modified and inserted back into the database.

        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@",
                                       "scott", "tiger");

        // It's faster when auto commit is off

```

```
conn.setAutoCommit (false);

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

try
{
    stmt.execute ("drop table student_table");
    stmt.execute ("drop type STUDENT");
}
catch (SQLException e)
{
    // the above drop and create statements will throw exceptions
    // if the types and tables did not exist before
}

stmt.execute ("create type STUDENT as object
              (name VARCHAR (30), age NUMBER)");
stmt.execute ("create table student_table of STUDENT");
stmt.execute ("insert into student_table values ('John', 20)");

ResultSet rs = stmt.executeQuery ("select ref (s) from student_table s");
rs.next ();

// retrieve the ref object
REF ref = (REF) rs.getObject (1);

//retrieve the object value that the ref points to in the
// object table

STRUCT student = (STRUCT) ref.getValue ();
Object attributes[] = student.getAttributes();

System.out.println ("student name: " + (String) attributes[0]);
System.out.println ("student age: " + ((BigDecimal)
                                attributes[1]).intValue());

rs.close();
stmt.close();
conn.close();
}
}
```


Weakly Typed Arrays—ArrayExample.java

This sample program uses JDBC to create a table with a VARRAY. It inserts a new array object into the table, then prints the contents of the table. For more information on arrays, see [Chapter 10, "Working with Oracle Collections"](#).

```
import java.sql.*;
import oracle.sql.*;
import oracle.jdbc.oracore.Util;
import oracle.jdbc.*;
import java.math.BigDecimal;

public class ArrayExample
{
    public static void main (String args[])
        throws Exception
    {
        // Register the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You need to put your database name after the @ sign in
        // the connection URL.
        //
        // The sample retrieves an varray of type "NUM_VARRAY",
        // materializes the object as an object of type ARRAY.
        // A new ARRAY is then inserted into the database.

        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@",
                                       "scott", "tiger");

        // It's faster when auto commit is off
        conn.setAutoCommit (false);

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

        try
        {
            stmt.execute ("DROP TABLE varray_table");
            stmt.execute ("DROP TYPE num_varray");
        }
        catch (SQLException e)
```

```
{
    // the above drop statements will throw exceptions
    // if the types and tables did not exist before. Just ignore it.
}

stmt.execute ("CREATE TYPE num_varray AS VARRAY(10) OF NUMBER(12, 2)");
stmt.execute ("CREATE TABLE varray_table (coll num_varray)");
stmt.execute ("INSERT INTO varray_table VALUES (num_varray(100, 200))");

ResultSet rs = stmt.executeQuery("SELECT * FROM varray_table");
showResultSet (rs);

//now insert a new row

// create a new ARRAY object
int elements[] = { 300, 400, 500, 600 };
ArrayDescriptor desc = ArrayDescriptor.createDescriptor("NUM_VARRAY", conn);
ARRAY newArray = new ARRAY(desc, conn, elements);

PreparedStatement ps =
    conn.prepareStatement ("insert into varray_table values (?)");
((OraclePreparedStatement)ps).setARRAY (1, newArray);

ps.execute ();

rs = stmt.executeQuery("SELECT * FROM varray_table");
showResultSet (rs);

// Close all the resources
rs.close();
ps.close();
stmt.close();
conn.close();

}

public static void showResultSet (ResultSet rs)
    throws SQLException
{
    int line = 0;
    while (rs.next())
    {
        line++;
        System.out.println("Row "+line+" : ");
        ARRAY array = ((OracleResultSet)rs).getARRAY (1);
    }
}
```

```
System.out.println ("Array is of type "+array.getSQLTypeName());
System.out.println
    ("Array element is of typecode "+array.getBaseType());
System.out.println ("Array is of length "+array.length());

// get Array elements
BigDecimal[] values = (BigDecimal[]) array.getArray();

for (int i=0; i<values.length; i++)
{
    BigDecimal value = (BigDecimal) values[i];
    System.out.println(">> index "+i+" = "+value.intValue());
}
}
}
```

Samples for Oracle Type Extensions

This section contains sample code for some of the Oracle type extensions:

- [REF CURSORS—RefCursorExample.java](#)
- [BFILES—FileExample.java](#)

The REF CURSOR sample is located in the following directory on the product CD:

```
[Oracle Home]/jdbc/demo/samples/oci8/basic-samples
```

The BFILE example is in the `object-samples` directory.

REF CURSORS—RefCursorExample.java

This sample program shows Oracle JDBC REF CURSOR functionality, creating a PL/SQL package that includes a stored function that returns a REF CURSOR type. The sample retrieves the REF CURSOR into a result set object. For information on REF CURSORS, see "[Oracle REF CURSOR Type Category](#)" on page 5-34.

```
/*
 * This sample shows how to call a PL/SQL function that opens
 * a cursor and get the cursor back as a Java ResultSet.
 */

import java.sql.*;
import java.io.*;
import oracle.jdbc.*;

class RefCursorExample
{
    public static void main (String args [])
        throws SQLException
    {
        // Load the driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        // Create the stored procedure
        init (conn);
    }
}
```

```

// Prepare a PL/SQL call
CallableStatement call =
    conn.prepareCall ("{ ? = call java_refcursor.job_listing (?)}");

// Find out all the SALES person
call.registerOutParameter (1, OracleTypes.CURSOR);
call.setString (2, "SALESMAN");
call.execute ();
ResultSet rset = (ResultSet)call.getObject (1);

// Dump the cursor
while (rset.next ())
    System.out.println (rset.getString ("ENAME"));

// Close all the resources
rset.close();
call.close();
conn.close();

}

// Utility function to create the stored procedure
static void init (Connection conn)
    throws SQLException
{
    Statement stmt = conn.createStatement ();

    stmt.execute ("create or replace package java_refcursor as " +
        " type myrctype is ref cursor return EMP%ROWTYPE; " +
        " function job_listing (j varchar2) return myrctype; " +
        "end java_refcursor;");

    stmt.execute ("create or replace package body java_refcursor as " +
        " function job_listing (j varchar2) return myrctype is " +
        " rc myrctype; " +
        " begin " +
        " open rc for select * from emp where job = j; " +
        " return rc; " +
        " end; " +
        "end java_refcursor;");
    stmt.close();
}
}

```

BFILES—FileExample.java

This sample demonstrates Oracle JDBC BFILE support. It illustrates filling a table with BFILES and includes a utility for dumping the contents of a BFILE. For information on BFILES, see ["Working with BFILES"](#) on page 7-20.

```
/*
 * This sample demonstrate basic File support
 */

import java.sql.*;
import java.io.*;
import java.util.*;

//including this import makes the code easier to read
import oracle.jdbc.*;

// needed for new BFILE class
import oracle.sql.*;

public class FileExample
{
    public static void main (String args [])
        throws Exception
    {
        // Register the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        //
        // The sample creates a DIRECTORY and you have to be connected as
        // "system" to be able to run the test.
        // I you can't connect as "system" have your system manager
        // create the directory for you, grant you the rights to it, and
        // remove the portion of this program that drops and creates the directory.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "system", "manager");

        // It's faster when auto commit is off
        conn.setAutoCommit (false);

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

```
try
{
    stmt.execute ("drop directory TEST_DIR");
}
catch (SQLException e)
{
    // An error is raised if the directory does not exist. Just ignore it.
}
stmt.execute ("create directory TEST_DIR as '/tmp/filetest'");

try
{
    stmt.execute ("drop table test_dir_table");
}
catch (SQLException e)
{
    // An error is raised if the table does not exist. Just ignore it.
}

// Create and populate a table with files
// The files file1 and file2 must exist in the directory TEST_DIR created
// above as symbolic name for /private/local/filetest.
stmt.execute ("create table test_dir_table (x varchar2 (30), b bfile)");
stmt.execute ("insert into test_dir_table values
              ('one', bfilename ('TEST_DIR', 'file1'))");
stmt.execute ("insert into test_dir_table values
              ('two', bfilename ('TEST_DIR', 'file2'))");

// Select the file from the table
ResultSet rset = stmt.executeQuery ("select * from test_dir_table");
while (rset.next ())
{
    String x = rset.getString (1);
    BFILE bfile = ((OracleResultSet)rset).getBFILE (2);
    System.out.println (x + " " + bfile);

    // Dump the file contents
    dumpBfile (conn, bfile);
}

// Close all resources
rset.close();
stmt.close();
conn.close();
}
```

```
// Utility function to dump the contents of a Bfile
static void dumpBfile (Connection conn, BFILE bfile)
    throws Exception
{
    System.out.println ("Dumping file " + bfile.getName());
    System.out.println ("File exists: " + bfile.fileExists());
    System.out.println ("File open: " + bfile.isFileOpen());

    System.out.println ("Opening File: ");

    bfile.openFile();

    System.out.println ("File open: " + bfile.isFileOpen());

    long length = bfile.length();
    System.out.println ("File length: " + length);

    int chunk = 10;

    InputStream instream = bfile.getBinaryStream();

    // Create temporary buffer for read
    byte[] buffer = new byte[chunk];

    // Fetch data
    while ((length = instream.read(buffer)) != -1)
    {
        System.out.print("Read " + length + " bytes: ");

        for (int i=0; i<length; i++)
            System.out.print(buffer[i]+" ");
        System.out.println();
    }

    // Close input stream
    instream.close();

    // close file handler
    bfile.closeFile();
}
}
```


Samples for Custom Object Classes

This section demonstrates the functionality of custom Java classes to map from SQL structured objects, providing examples of both a standard `SQLData` implementation and an Oracle `ORADData` implementation:

- [SQLData Implementation—SQLDataExample.java](#)
- [ORADData Implementation—ORADDataExample.java](#)

This includes examples of the code you must provide to define custom Java classes for Oracle objects, and sample applications that make use of these custom Java class definitions. You create the custom classes by implementing either the standard `java.sql.SQLData` interface or the Oracle `oracle.sql.CustomDatum` interface. These interfaces provide a way to create and populate the custom Java class for the Oracle object and its attributes.

`SQLData` and `CustomDatum` both populate a Java object from a SQL object, with the `SQLData` interface providing more portability and the `CustomDatum` interface providing more utility and flexibility in how you present the data.

The `SQLData` interface is a JDBC standard. For more information on this interface, see ["Understanding the SQLData Interface"](#) on page 8-15.

The `CustomDatum` interface is provided by Oracle. For more information on the `CustomDatum` interface, see ["Understanding the ORADData Interface"](#) on page 8-21.

You can write your own code to create custom Java classes that implement either interface, but the Oracle `JPublisher` utility can generate classes to implement either interface as well.

For more information about `JPublisher`, see ["Using JPublisher to Create Custom Object Classes"](#) on page 8-45 and the *Oracle9i JPublisher User's Guide*.

The sample applications and custom Java class definitions in this section are located in the following directory on the product CD:

```
[Oracle Home]/jdbc/demo/samples/oci8/object-samples
```

SQLData Implementation—SQLDataExample.java

This section contains code that illustrates how to define and use a custom Java type corresponding to a given SQL object type, using a `SQLData` implementation.

SQL Object Definition

Following is the SQL definition of an `EMPLOYEE` object. The object has two attributes: a `VARCHAR2` attribute `EMPNAME` (employee name) and an `INTEGER` attribute `EMPNO` (employee number).

```
-- SQL definition
CREATE TYPE employee AS OBJECT
(
    empname VARCHAR2(50),
    empno   INTEGER
);
```

Custom Object Class—SQLData Implementation

The following code defines the custom Java class `EmployeeObj` (defined in `EmployeeObj.java`) to correspond to the SQL type `EMPLOYEE`. Notice that the definition of `EmployeeObj` contains a string `empName` (employee name) attribute and an integer `empNo` (employee number) attribute. Also notice that the Java definition of the `EmployeeObj` custom Java class implements the `SQLData` interface and includes the implementations of a `get` method and the required `readSQL()` and `writeSQL()` methods.

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

public class EmployeeObj implements SQLData
{
    private String sql_type;

    public String empName;
    public int empNo;

    public EmployeeObj()
    {
    }

    public EmployeeObj (String sql_type, String empName, int empNo)
    {
        this.sql_type = sql_type;
        this.empName = empName;
        this.empNo = empNo;
    }

    /////// implements SQLData /////
}
```

```
public String getSQLTypeName() throws SQLException
{
    return sql_type;
}

public void readSQL(SQLInput stream, String typeName)
    throws SQLException
{
    sql_type = typeName;

    empName = stream.readString();
    empNo = stream.readInt();
}

public void writeSQL(SQLOutput stream)
    throws SQLException
{
    stream.writeString(empName);
    stream.writeInt(empNo);
}
}
```

Sample Application Using SQLData Custom Object Class

After you create the `EmployeeObj` Java class, you can use it in a program. The following program creates a table that stores employee name and number data. The program uses the `EmployeeObj` object to create a new employee object and insert it in the table. It then applies a `SELECT` statement to get the contents of the table and prints its contents.

For information about using `SQLData` implementations to access and manipulate SQL object data, see "[Reading and Writing Data with a SQLData Implementation](#)" on page 8-17.

```
import java.sql.*;
import oracle.jdbc.*;
import oracle.sql.*;
import java.math.BigDecimal;
import java.util.Dictionary;

public class SQLDataExample
{
```

```
public static void main(String args []) throws Exception
{
    // Connect
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver ());
    OracleConnection conn = (OracleConnection)
        DriverManager.getConnection("jdbc:oracle:oci8:@",
            "scott", "tiger");

    Dictionary map = (Dictionary)conn.getTypeMap();
    map.put("EMPLOYEE", Class.forName("EmployeeObj"));

    // Create a Statement
    Statement stmt = conn.createStatement ();
    try
    {
        stmt.execute ("drop table EMPLOYEE_TABLE");
        stmt.execute ("drop type EMPLOYEE");
    }
    catch (SQLException e)
    {
        // An error is raised if the table/type does not exist. Just ignore it.
    }

    // Create and populate tables
    stmt.execute ("CREATE TYPE EMPLOYEE AS OBJECT
        (EmpName VARCHAR2(50),EmpNo INTEGER)");
    stmt.execute ("CREATE TABLE EMPLOYEE_TABLE (ATTR1 EMPLOYEE)");
    stmt.execute ("INSERT INTO EMPLOYEE_TABLE VALUES
        (EMPLOYEE('Susan Smith', 123))");
    stmt.close();

    // Create a SQLData object
    EmployeeObj e = new EmployeeObj("SCOTT.EMPLOYEE", "George Jones", 456);

    // Insert the SQLData object
    PreparedStatement pstmt
        = conn.prepareStatement ("insert into employee_table values (?)");

    pstmt.setObject(1, e, OracleTypes.STRUCT);
    pstmt.executeQuery();
    System.out.println("insert done");
    pstmt.close();
}
```

```

// Select now
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)
    s.executeQuery("select * from employee_table");

while(rs.next())
{
    EmployeeObj ee = (EmployeeObj) rs.getObject(1);
    System.out.println("EmpName: " + ee.empName + " EmpNo: " + ee.empNo);
}
rs.close();
s.close();

if (conn != null)
{
    conn.close();
}
}
}

```

ORADData Implementation—ORADDataExample.java

This section contains code that illustrates how to define and use a custom Java type corresponding to a given SQL object type, using a `ORADData` implementation.

SQL Object Definition

Following is the SQL definition of an `EMPLOYEE` object. The object has two attributes: a `VARCHAR2` attribute `EMPNAME` (employee name) and an `INTEGER` attribute `EMPNO` (employee number).

```

CREATE TYPE employee AS OBJECT
(
    empname VARCHAR2(50),
    empno   INTEGER
);

```

Custom Object Class—CustomDatum Implementation

The following code defines the custom Java class `Employee` (defined in `Employee.java`) to correspond to the SQL type `EMPLOYEE`. Notice that the definition of `Employee` contains accessor methods for a string `empname` (employee name) and an integer `empno` (employee number). Also notice that the Java definition of the `Employee` custom Java class implements the `ORADData` and

ORADatafactory interfaces. A custom Java class that implements ORADatafactory has a static getFactory() method that returns a ORADatafactory object. The JDBC driver uses the ORADatafactory object's create() method to return a CustomDatum instance.

Note that instead of writing the custom Java class yourself, you can use the JPublisher utility to generate class definitions that implement the ORADatafactory and ORADatafactory interfaces. In fact, the Employee.java code shown here was generated by JPublisher.

```
import java.sql.SQLException;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Employee implements CustomDatum, CustomDatumFactory
{
    public static final String _SQL_NAME = "SCOTT.EMPLOYEE";
    public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

    MutableStruct _struct;

    static int[] _sqlType =
    {
        12, 4
    };

    static CustomDatumFactory[] _factory = new CustomDatumFactory[2];

    static final Employee _EmployeeFactory = new Employee();
    public static CustomDatumFactory getFactory()
    {
        return _EmployeeFactory;
    }

    /* constructor */
    public Employee()
    {
        _struct = new MutableStruct(new Object[2], _sqlType, _factory);
    }
}
```

```

/* CustomDatum interface */
public Datum toDatum(OracleConnection c) throws SQLException
{
    return _struct.toDatum(c, _SQL_NAME);
}

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
    if (d == null) return null;
    Employee o = new Employee();
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
    return o;
}

/* accessor methods */
public String getEmpname() throws SQLException
{ return (String) _struct.getAttribute(0); }

public void setEmpname(String empname) throws SQLException
{ _struct.setAttribute(0, empname); }

public Integer getEmpno() throws SQLException
{ return (Integer) _struct.getAttribute(1); }

public void setEmpno(Integer empno) throws SQLException
{ _struct.setAttribute(1, empno); }
}

```

Sample Application Using CustomDatum Custom Object Class

This sample program shows how you can use the `Employee` class generated by `JPublisher`. The sample code creates a new `Employee` object, fills it with data, then inserts it into the database. The sample code then retrieves the `Employee` data from the database.

For information about using `CustomDatum` implementations to access and manipulate SQL object data, see ["Reading and Writing Data with a ORADData Implementation"](#) on page 8-23.

```

import java.sql.*;
import oracle.jdbc.*;
import oracle.sql.*;
import java.math.BigDecimal;

```

```
public class CustomDatumExample
{

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

        // Connect
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver ());
        OracleConnection conn = (OracleConnection)
            DriverManager.getConnection("jdbc:oracle:oci8:@",
                                     "scott", "tiger");

        // Create a Statement
        Statement stmt = conn.createStatement ();
        try
        {
            stmt.execute ("drop table EMPLOYEE_TABLE");
            stmt.execute ("drop type EMPLOYEE");
        }
        catch (SQLException e)
        {
            // An error is raised if the table/type does not exist. Just ignore it.
        }

        // Create and populate tables
        stmt.execute ("CREATE TYPE EMPLOYEE AS OBJECT
                     (EmpName VARCHAR2(50),EmpNo INTEGER)");
        stmt.execute ("CREATE TABLE EMPLOYEE_TABLE (ATTR1 EMPLOYEE)");
        stmt.execute ("INSERT INTO EMPLOYEE_TABLE VALUES
                     (EMPLOYEE('Susan Smith', 123))");
        stmt.close();

        // Create a CustomDatum object
        Employee e = new Employee("George Jones", new BigDecimal("456"));

        // Insert the CustomDatum object
        PreparedStatement pstmt
            = conn.prepareStatement ("insert into employee_table values (?)");

        pstmt.setObject(1, e, OracleTypes.STRUCT);
        pstmt.executeQuery();
        System.out.println("insert done");
        pstmt.close();
    }
}
```



```
// Select now
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)
    s.executeQuery("select * from employee_table");

while(rs.next())
{
    Employee ee = (Employee) rs.getCustomDatum(1, Employee.getFactory());
    System.out.println("EmpName: " + ee.empName + " EmpNo: " + ee.empNo);
}
rs.close();
s.close();

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

JDBC 2.0 Result Set Enhancement Samples

This section provides samples that demonstrate the functionality of result set enhancements available with JDBC 2.0. This includes positioning in a scrollable result set, updating a result set, using a scroll-sensitive result set that can automatically see external updates, and explicitly refetching data into a result set:

- [Positioning in a Result Set—ResultSet2.java](#)
- [Inserting and Deleting Rows in a Result Set—ResultSet3.java](#)
- [Updating Rows in a Result Set—ResultSet4.java](#)
- [Scroll-Sensitive Result Set—ResultSet5.java](#)
- [Refetching Rows in a Result Set—ResultSet6.java](#)

The sample applications in this section are located in the following directory on the product CD:

```
[Oracle Home]/jdbc/demo/samples/oci8/jdbc20-samples
```

Positioning in a Result Set—ResultSet2.java

This section demonstrates scrollable result set functionality—moving to relative and absolute row positions and iterating backwards through the result set.

For discussion on these topics, see "[Positioning and Processing in Scrollable Result Sets](#)" on page 11-13.

```
/**
 * A simple sample to demonstrate previous(), absolute() and relative().
 */

import java.sql.*;

public class ResultSet2
{
    public static void main(String[] args) throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");
```

```
// Create a Statement
Statement stmt = conn.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,
                                       ResultSet.CONCUR_UPDATABLE);

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

// iterate through the result using next()
show_resultset_by_next(rset);

// iterate through the result using previous()
show_resultset_by_previous(rset);

// iterate through the result using absolute()
show_resultset_by_absolute(rset);

// iterate through the result using relative()
show_resultset_by_relative(rset);

// Close the ResultSet
rset.close();

// Close the Statement
stmt.close();

// Close the connection
conn.close();
}

/**
 * Iterate through the result using next().
 *
 * @param rset a result set object
 */
public static void show_resultset_by_next(ResultSet rset)
    throws SQLException
{
    System.out.println ("List the employee names using ResultSet.next():");

    // Make sure the cursor is placed right before the first row
    if (!rset.isBeforeFirst())
    {
        // Place the cursor right before the first row
        rset.beforeFirst ();
    }
}
```

```

    }

    // Iterate through the rows using next()
    while (rset.next())
        System.out.println (rset.getString (1));

    System.out.println ();
}

/**
 * Iterate through the result using previous().
 *
 * @param rset a result set object
 */
public static void show_resultset_by_previous(ResultSet rset)
    throws SQLException
{
    System.out.println ("List the employee names using ResultSet.previous():");

    // Make sure the cursor is placed after the last row
    if (!rset.isAfterLast())
    {
        // Place the cursor after the last row
        rset.afterLast ();
    }

    // Iterate through the rows using previous()
    while (rset.previous())
        System.out.println (rset.getString (1));

    System.out.println ();
}

/**
 * Iterate through the result using absolute().
 *
 * @param rset a result set object
 */
public static void show_resultset_by_absolute (ResultSet rset)
    throws SQLException
{
    System.out.println ("List the employee names using ResultSet.absolute():");

    // The begin index for ResultSet.absolute (idx)
    int idx = 1;

```

```
// Loop through the result set until absolute() returns false.
while (rset.absolute(idx))
{
    System.out.println (rset.getString (1));
    idx ++;
}
System.out.println ();
}

/**
 * Iterate through the result using relative().
 *
 * @param rset a result set object
 */
public static void show_resultset_by_relative (ResultSet rset)
    throws SQLException
{
    System.out.println ("List the employee names using ResultSet.relative():");

    // getRow() returns 0 if there is no current row
    if (rset.getRow () == 0 || !rset.isLast())
    {
        // place the cursor on the last row
        rset.last ();
    }

    // Calling relative(-1) is similar to previous(), but the cursor
    // has to be on a valid row before calling relative().
    do
    {
        System.out.println (rset.getString (1));
    }
    while (rset.relative (-1));

    System.out.println ();
}
}
```

Inserting and Deleting Rows in a Result Set—ResultSet3.java

This sample shows some of the functionality of an updatable result set—inserting and deleting rows that will in turn be inserted into or deleted from the database.

For discussion on these topics, see ["Performing an INSERT Operation in a Result Set"](#) on page 11-21 and ["Performing a DELETE Operation in a Result Set"](#) on page 11-18.

```
/**
 * A simple sample to demonstrate ResultSet.insertRow() and
 * ResultSet.deleteRow().
 */

import java.sql.*;

public class ResultSet3
{
    public static void main(String[] args) throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        // Cleanup
        cleanup (conn);

        // Create a Statement
        Statement stmt = conn.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                ResultSet.CONCUR_UPDATABLE);

        // Query the EMP table
        ResultSet rset = stmt.executeQuery ("select EMPNO, ENAME from EMP");

        // Add three new employees using ResultSet.insertRow()
        addEmployee (rset, 1001, "PETER");
        addEmployee (rset, 1002, "MARY");
        addEmployee (rset, 1003, "DAVID");

        // Close the result set
        rset.close ();

        // Verify the insertion
        System.out.println ("\nList EMPNO and ENAME in the EMP table: ");

        rset = stmt.executeQuery ("select EMPNO, ENAME from EMP");
    }
}
```

```
while (rset.next())
{
    // We expect to see the three new employees
    System.out.println (rset.getInt(1)+" "+rset.getString(2));
}
System.out.println ();

// Delete the new employee 'PETER' using ResultSet.deleteRow()
removeEmployee (rset, 1001);
rset.close ();

// Verify the deletion
System.out.println ("\nList EMPNO and ENAME in the EMP table: ");
rset = stmt.executeQuery ("select EMPNO, ENAME from EMP");
while (rset.next())
{
    // We expect "PETER" is removed
    System.out.println (rset.getInt(1)+" "+rset.getString(2));
}
System.out.println ();

// Close the RresultSet
rset.close();

// Close the Statement
stmt.close();

// Cleanup
cleanup(conn);

// Close the connection
conn.close();
}

/**
 * Add a new employee to EMP table.
 */
public static void addEmployee (ResultSet rset,
                                int employeeId,
                                String employeeName)
    throws SQLException
{
    System.out.println ("Adding new employee: "+employeeId+" "+employeeName);

    // Place the cursor on the insert row
```

```
        rset.moveToInsertRow();

        // Assign the new values
        rset.updateInt (1, employeeId);
        rset.updateString (2, employeeName);

        // Insert the new row to database
        rset.insertRow();
    }

    /**
     * Remove the employee from EMP table.
     */
    public static void removeEmployee (ResultSet rset,
                                       int employeeId)
        throws SQLException
    {
        System.out.println ("Removing the employee: id="+employeeId);

        // Place the cursor right before the first row if it doesn't
        if (!rset.isBeforeFirst())
        {
            rset.beforeFirst();
        }

        // Iterate the result set
        while (rset.next())
        {
            // Place the cursor the row with matched employee id
            if (rset.getInt(1) == employeeId)
            {
                // Delete the current row
                rset.deleteRow();
                break;
            }
        }
    }

    /**
     * Generic cleanup.
     */
    public static void cleanup (Connection conn)
        throws SQLException
    {
        Statement stmt = conn.createStatement ();
```



```

        stmt.execute
            ("DELETE FROM EMP WHERE EMPNO=1001 OR EMPNO=1002 OR EMPNO=1003");
        stmt.execute ("COMMIT");
        stmt.close ();
    }
}

```

Updating Rows in a Result Set—ResultSet4.java

This sample shows some of the functionality of an updatable result set—updating rows that will in turn be updated in the database.

For a discussion on this topic, see ["Performing an UPDATE Operation in a Result Set"](#) on page 11-19.

```

/**
 * A simple sample to demonstrate ResultSet.updateRow().
 */

import java.sql.*;

public class ResultSet4
{
    public static void main(String[] args) throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        // Create a Statement
        Statement stmt = conn.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                ResultSet.CONCUR_UPDATABLE);

        // Query the EMP table
        ResultSet rset = stmt.executeQuery ("select EMPNO, ENAME, SAL from EMP");

        // Give everybody a $500 raise
        adjustSalary (rset, 500);

        // Verify the salary changes
    }
}

```

```
System.out.println ("Verify the changes with a new query: ");
rset = stmt.executeQuery ("select EMPNO, ENAME, SAL from EMP");
while (rset.next())
{
    System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
                        rset.getInt(3));
}
System.out.println ();

// Close the RresultSet
rset.close();

// Close the Statement
stmt.close();

// Cleanup
cleanup(conn);

// Close the connection
conn.close();
}

/**
 * Update the ResultSet content using updateRow().
 */
public static void adjustSalary (ResultSet rset, int raise)
    throws SQLException
{
    System.out.println ("Give everybody in the EMP table a $500 raise\n");

    int salary = 0;

    while (rset.next ())
    {
        // save the old value
        salary = rset.getInt (3);

        // update the row
        rset.updateInt (3, salary + raise);

        // flush the changes to database
        rset.updateRow ();

        // show the changes
        System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
```

```

        salary+" -> "+rset.getInt(3));
    }
    System.out.println ();
}

/**
 * Generic cleanup.
 */
public static void cleanup (Connection conn) throws SQLException
{
    Statement stmt = conn.createStatement ();
    stmt.execute ("UPDATE EMP SET SAL = SAL - 500");
    stmt.execute ("COMMIT");
    stmt.close ();
}
}

```

Scroll-Sensitive Result Set—ResultSet5.java

This sample shows the functionality of a scroll-sensitive result. Such a result set can implicitly see updates to the database that were made externally.

For more information about scroll-sensitive result sets and how they are implemented, see ["Oracle Implementation of Scroll-Sensitive Result Sets"](#) on page 11-30.

```

/**
 * A simple sample to demonstrate scroll sensitive result set.
 */

import java.sql.*;

public class ResultSet5
{
    public static void main(String[] args) throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");
    }
}

```

```
// Create a Statement
Statement stmt = conn.createStatement (ResultSet.TYPE_SCROLL_SENSITIVE,
                                       ResultSet.CONCUR_UPDATABLE);

// Set the statement fetch size to 1
stmt.setFetchSize (1);

// Query the EMP table
ResultSet rset = stmt.executeQuery ("select EMPNO, ENAME, SAL from EMP");

// List the result set's type, concurrency type, ..., etc
showProperty (rset);

// List the query result
System.out.println ("List ENO, ENAME and SAL from the EMP table: ");
while (rset.next())
{
    System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
                       rset.getInt(3));
}
System.out.println ();

// Do some changes outside the result set
doSomeChanges (conn);

// Place the cursor right before the first row
rset.beforeFirst ();

// List the employee information again
System.out.println ("List ENO, ENAME and SAL again: ");
while (rset.next())
{
    // We expect to see the changes made in "doSomeChanges()"
    System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
                       rset.getInt(3));
}

// Close the ResultSet
rset.close();

// Close the Statement
stmt.close();

// Cleanup
cleanup(conn);
```

```
        // Close the connection
        conn.close();
    }

    /**
     * Update the EMP table.
     */
    public static void doSomeChanges (Connection conn)
        throws SQLException
    {
        System.out.println ("Update the employee salary outside the result set\n");

        Statement otherStmt = conn.createStatement ();
        otherStmt.execute ("update emp set sal = sal + 500");
        otherStmt.execute ("commit");
        otherStmt.close ();
    }

    /**
     * Show the result set properties like type, concurrency type, fetch
     * size, ..., etc.
     */
    public static void showProperty (ResultSet rset) throws SQLException
    {
        // Verify the result set type
        switch (rset.getType())
        {
            case ResultSet.TYPE_FORWARD_ONLY:
                System.out.println ("Result set type: TYPE_FORWARD_ONLY");
                break;
            case ResultSet.TYPE_SCROLL_INSENSITIVE:
                System.out.println ("Result set type: TYPE_SCROLL_INSENSITIVE");
                break;
            case ResultSet.TYPE_SCROLL_SENSITIVE:
                System.out.println ("Result set type: TYPE_SCROLL_SENSITIVE");
                break;
            default:
                System.out.println ("Invalid type");
                break;
        }

        // Verify the result set concurrency
        switch (rset.getConcurrency())
        {
```

```
        case ResultSet.CONCUR_UPDATABLE:
            System.out.println
                ("Result set concurrency: ResultSet.CONCUR_UPDATABLE");
            break;
        case ResultSet.CONCUR_READ_ONLY:
            System.out.println
                ("Result set concurrency: ResultSet.CONCUR_READ_ONLY");
            break;
        default:
            System.out.println ("Invalid type");
            break;
    }

    // Verify the fetch size
    System.out.println ("fetch size: "+rset.getFetchSize ());

    System.out.println ();
}

/**
 * Generic cleanup.
 */
public static void cleanup (Connection conn) throws SQLException
{
    Statement stmt = conn.createStatement ();
    stmt.execute ("UPDATE EMP SET SAL = SAL - 500");
    stmt.execute ("COMMIT");
    stmt.close ();
}
}
```

Refetching Rows in a Result Set—ResultSet6.java

This sample shows how to explicitly refetch data from the database to update the result set. This functionality is available in scroll-sensitive and scroll-insensitive/updatable result sets.

For more information, see ["Refetching Rows"](#) on page 11-26.

```
/**
 * A simple sample to demonstrate ResultSet.refreshRow().
 */

import java.sql.*;
```

```
public class ResultSet6
{
    public static void main(String[] args) throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        // Create a Statement
        Statement stmt = conn.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                ResultSet.CONCUR_UPDATABLE);

        // Set the statement fetch size to 1
        stmt.setFetchSize (1);

        // Query the EMP table
        ResultSet rset = stmt.executeQuery ("select EMPNO, ENAME, SAL from EMP");

        // List the result set's type, concurrency type, ..., etc
        showProperty (rset);

        // List the query result
        System.out.println ("List ENO, ENAME and SAL from the EMP table: ");
        while (rset.next())
        {
            System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
                                rset.getInt(3));
        }
        System.out.println ();

        // Do some changes outside the result set
        doSomeChanges (conn);

        // Place the cursor right before the first row
        rset.beforeFirst ();

        // List the employee information again
        System.out.println ("List ENO, ENAME and SAL again: ");
        int salary = 0;
        while (rset.next())
```

```
{
    // save the original salary
    salary = rset.getInt (3);

    // refresh the row
    rset.refreshRow ();

    // We expect to see the changes made in "doSomeChanges()"
    System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
        salary+" -> "+rset.getInt(3));
}

// Close the ResultSet
rset.close();

// Close the Statement
stmt.close();

// Cleanup
cleanup(conn);

// Close the connection
conn.close();
}

/**
 * Update the EMP table.
 */
public static void doSomeChanges (Connection conn)
    throws SQLException
{
    System.out.println ("Update the employee salary outside the result set\n");

    Statement otherStmt = conn.createStatement ();
    otherStmt.execute ("update emp set sal = sal + 500");
    otherStmt.execute ("commit");
    otherStmt.close ();
}

/**
 * Show the result set properties like type, concurrency type, fetch
 * size, ..., etc.
 */
public static void showProperty (ResultSet rset) throws SQLException
{
```



```
// Verify the result set type
switch (rset.getType())
{
    case ResultSet.TYPE_FORWARD_ONLY:
        System.out.println ("Result set type: TYPE_FORWARD_ONLY");
        break;
    case ResultSet.TYPE_SCROLL_INSENSITIVE:
        System.out.println ("Result set type: TYPE_SCROLL_INSENSITIVE");
        break;
    case ResultSet.TYPE_SCROLL_SENSITIVE:
        System.out.println ("Result set type: TYPE_SCROLL_SENSITIVE");
        break;
    default:
        System.out.println ("Invalid type");
        break;
}

// Verify the result set concurrency
switch (rset.getConcurrency())
{
    case ResultSet.CONCUR_UPDATABLE:
        System.out.println
            ("Result set concurrency: ResultSet.CONCUR_UPDATABLE");
        break;
    case ResultSet.CONCUR_READ_ONLY:
        System.out.println
            ("Result set concurrency: ResultSet.CONCUR_READ_ONLY");
        break;
    default:
        System.out.println ("Invalid type");
        break;
}

// Verify the fetch size
System.out.println ("fetch size: "+rset.getFetchSize ());

System.out.println ();
}

/**
 * Generic cleanup.
 */
public static void cleanup (Connection conn) throws SQLException
{
    Statement stmt = conn.createStatement ();
```

```
        stmt.execute ("UPDATE EMP SET SAL = SAL - 500");
        stmt.execute ("COMMIT");
        stmt.close ();
    }
}
```

Performance Enhancement Samples

This section provides sample applications for performance enhancement features such as update batching:

- [Standard Update Batching—BatchUpdates.java](#)
- [Oracle Update Batching with Implicit Execution—SetExecuteBatch.java](#)
- [Oracle Update Batching with Explicit Execution—SendBatch.java](#)
- [Oracle Row Prefetching Specified in Connection—RowPrefetch_connection.java](#)
- [Oracle Row Prefetching Specified in Statement—RowPrefetch_statement.java](#)
- [Oracle Column Type Definitions—DefineColumnType.java](#)

The sample applications for Oracle-specific performance enhancements are located in the following directory on the product CD:

```
[Oracle Home]/jdbc/demo/samples/oci8/basic-samples
```

The standard update batching sample is located in the `jdbc20-samples` directory.

Standard Update Batching—BatchUpdates.java

This sample shows how to use standard update batching as specified by JDBC 2.0. For more information, see "[Standard Update Batching](#)" on page 12-10.

For comparison and contrast between the standard and Oracle-specific update batching models, see "[Overview of Update Batching Models](#)" on page 12-2.

```
/**
 * A simple sample to demonstrate standard JDBC 2.0 update batching.
 */

import java.sql.*;

public class BatchUpdates
{
    public static void main(String[] args)
    {
        Connection        conn = null;
        Statement          stmt = null;
        PreparedStatement   pstmt = null;
        ResultSet          rset = null;
        int                i = 0;
```

```
try
{
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    conn = DriverManager.getConnection(
        "jdbc:oracle:oci8:@", "scott", "tiger");

    stmt = conn.createStatement();
    try { stmt.execute(
        "create table mytest_table (col1 number, col2 varchar2(20))");
    } catch (Exception e1) {}

    //
    // Insert in a batch.
    //
    pstmt = conn.prepareStatement("insert into mytest_table values (?, ?)");

    pstmt.setInt(1, 1);
    pstmt.setString(2, "row 1");
    pstmt.addBatch();

    pstmt.setInt(1, 2);
    pstmt.setString(2, "row 2");
    pstmt.addBatch();

    pstmt.executeBatch();

    //
    // Select and print results.
    //
    rset = stmt.executeQuery("select * from mytest_table");
    while (rset.next())
    {
        System.out.println(rset.getInt(1) + ", " + rset.getString(2));
    }
}
catch (Exception e)
{
    e.printStackTrace();
}
finally
{
    if (stmt != null)
    {
        try { stmt.execute("drop table mytest_table"); } catch (Exception e) {}
    }
}
```

```

        try { stmt.close(); } catch (Exception e) {}
    }
    if (pstmt != null)
    {
        try { pstmt.close(); } catch (Exception e) {}
    }
    if (conn != null)
    {
        try { conn.close(); } catch (Exception e) {}
    }
}
}
}

```

Oracle Update Batching with Implicit Execution—SetExecuteBatch.java

This sample shows how to use Oracle update batching, with the batch being executed implicitly when the batch value (the number of statements to collect before sending them to the database) is reached.

For information about Oracle update batching, see ["Oracle Update Batching"](#) on page 12-4.

For comparison and contrast between the standard and Oracle-specific update batching models, see ["Overview of Update Batching Models"](#) on page 12-2.

```

/*
 * This sample shows how to use the batching extensions.
 * In this example, we set the defaultBatch value from the
 * connection object. This affects all statements created from
 * this connection.
 * It is possible to set the batch value individually for each
 * statement. The API to use on the statement object is setExecuteBatch().
 *
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// You need to import oracle.jdbc.* in order to use the
// API extensions.
import oracle.jdbc.*;

class SetExecuteBatch

```

```
{
public static void main (String args [])
    throws SQLException
{
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
        DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Default batch value set to 2 for all prepared statements belonging
    // to this connection.
    ((OracleConnection)conn).setDefaultExecuteBatch (2);

    PreparedStatement ps =
        conn.prepareStatement ("insert into dept values (?, ?, ?)");

    ps.setInt (1, 12);
    ps.setString (2, "Oracle");
    ps.setString (3, "USA");

    // No data is sent to the database by this call to executeUpdate
    System.out.println ("Number of rows updated so far: "
        + ps.executeUpdate ());

    ps.setInt (1, 11);
    ps.setString (2, "Applications");
    ps.setString (3, "Indonesia");

    // The number of batch calls to executeUpdate is now equal to the
    // batch value of 2. The data is now sent to the database and
    // both rows are inserted in a single roundtrip.
    int rows = ps.executeUpdate ();
    System.out.println ("Number of rows updated now: " + rows);

    ps.close ();
    conn.close();
}
}
```

Oracle Update Batching with Explicit Execution—SendBatch.java

This sample shows how to use Oracle update batching, with the batch being executed explicitly with a `sendBatch()` call.

For information about Oracle update batching, see ["Oracle Update Batching"](#) on page 12-4.

For comparison and contrast between the standard and Oracle-specific update batching models, see ["Overview of Update Batching Models"](#) on page 12-2.

```
/*
 * This sample shows how to use the batching extensions.
 * In this example, we demonstrate the use of the "sendBatch" API.
 * This allows the user to actually execute a set of batched
 * execute commands.
 *
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// You need to import oracle.jdbc.* in order to use the
// API extensions.
import oracle.jdbc.*;

class SendBatch
{
    public static void main (String args [])
        throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        Statement stmt = conn.createStatement ();

        // Default batch value set to 50 for all prepared statements belonging
        // to this connection.
        ((OracleConnection)conn).setDefaultExecuteBatch (50);

        PreparedStatement ps =
```

```
        conn.prepareStatement ("insert into dept values (?, ?, ?)");

        ps.setInt (1, 32);
        ps.setString (2, "Oracle");
        ps.setString (3, "USA");

        // this execute does not actually happen at this point
        System.out.println (ps.executeUpdate ());

        ps.setInt (1, 33);
        ps.setString (2, "Applications");
        ps.setString (3, "Indonesia");

        // this execute does not actually happen at this point
        int rows = ps.executeUpdate ();

        System.out.println ("Number of rows updated before calling sendBatch: "
            + 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 ();
        conn.close ();
    }
}
```

Oracle Row Prefetching Specified in Connection—RowPrefetch_connection.java

This section demonstrates how to use Oracle row prefetching-functionality, setting the row prefetch value in the connection object and thereby affecting every statement produced from that connection.

Note that Oracle row prefetching is fundamentally similar to JDBC 2.0 fetch size functionality.

For information about Oracle row prefetching, see ["Oracle Row Prefetching"](#) on page 12-20. For information about JDBC 2.0 fetch size and some comparison with row prefetching, see ["Fetch Size"](#) on page 11-24.


```
/*
 * This sample shows how to use the Oracle performance extensions
 * for row-prefetching. This allows the driver to fetch multiple
 * rows in one round-trip, saving unnecessary round-trips to the database.
 *
 * This example shows how to set the rowPrefetch for the connection object,
 * which will be used for all statements created from this connection.
 * Please see RowPrefetch_statement.java for examples of how to set
 * the rowPrefetch for statements individually.
 *
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// You need to import oracle.jdbc.driver in order to use the oracle extensions.
import oracle.jdbc.*;

class RowPrefetch_connection
{
    public static void main (String args [])
        throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        // set the RowPrefetch value from the Connection object
        // This sets the rowPrefetch for *all* statements belonging
        // to this connection.
        // The rowPrefetch value can be overridden for specific statements by
        // using the setRowPrefetch API on the statement object. Please look
        // at RowPrefetch_statement.java for an example.

        // Please note that any statements created *before* the connection
        // rowPrefetch was set, will use the default rowPrefetch.

        ((OracleConnection)conn).setDefaultRowPrefetch (30);

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

```
// Check to verify statement rowPrefetch value is 30.
int row_prefetch = ((OracleStatement)stmt).getRowPrefetch ();
System.out.println ("The RowPrefetch for the statement is: "
                    + row_prefetch + "\n");

ResultSet rset = stmt.executeQuery ("select ename from emp");

while(rset.next ())
{
    System.out.println (rset.getString (1));
}
rset.close ();
stmt.close ();
conn.close ();
}
}
```

Oracle Row Prefetching Specified in Statement—RowPrefetch_statement.java

This section demonstrates how to use Oracle row prefetching functionality, setting the row prefetch value in a particular statement object to override the value in the connection object producing the statement.

Note that Oracle row prefetching is fundamentally similar to JDBC 2.0 fetch size functionality.

For information about Oracle row prefetching, see ["Oracle Row Prefetching"](#) on page 12-20. For information about JDBC 2.0 fetch size and some comparison with row prefetching, see ["Fetch Size"](#) on page 11-24.

```
/*
 * This sample shows how to use the Oracle performance extensions
 * for row-prefetching. This allows the driver to fetch multiple
 * rows in one round-trip, saving unnecessary round-trips to the database.
 *
 * This example shows how to set the rowPrefetch for individual
 * statements.
 *
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;
```

```
// You need to import oracle.jdbc in order to use the
// Oracle extensions
import oracle.jdbc.*;

class RowPrefetch_statement
{
    public static void main (String args [])
        throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        // get the value of the default row prefetch from the connection object

        int default_row_prefetch =
            ((OracleConnection)conn).getDefaultRowPrefetch ();
        System.out.println ("The Default RowPrefetch for the connection is: "
            + default_row_prefetch);

        Statement stmt = conn.createStatement ();

        // set the RowPrefetch value from the statement object
        // This sets the rowPrefetch only for this particular statement.
        // All other statements will use the default RowPrefetch from the
        // connection.

        ((OracleStatement)stmt).setRowPrefetch (30);

        // Check to verify statement rowPrefetch value is 30.
        int row_prefetch = ((OracleStatement)stmt).getRowPrefetch ();
        System.out.println ("The RowPrefetch for the statement is: "
            + row_prefetch + "\n");

        ResultSet rset = stmt.executeQuery ("select ename from emp");

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

```
        stmt.close ();
        stmt.close ();
    }
}
```

Oracle Column Type Definitions—DefineColumnType.java

This sample shows how to use Oracle extensions to predefine result set column types to reduce round trips to the database for a query.

For information about column type definitions, see ["Defining Column Types"](#) on page 12-23.

```
/*
 * This sample shows how to use the "define" extensions.
 * The define extensions allow the user to specify the types
 * under which to retrieve column data in a query.
 *
 * This saves round-trips to the database (otherwise necessary to
 * gather information regarding the types in the select-list) and
 * conversions from native types to the types under which the user
 * will get the data.
 *
 * This can also be used to avoid streaming of long columns, by defining
 * them as CHAR or VARCHAR types.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// You need to import oracle.jdbc.* in order to use the
// API extensions.
import oracle.jdbc.*;

class DefineColumnType
{
    public static void main (String args [])
        throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver( new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
    }
}
```

```
Connection conn =
    DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

Statement stmt = conn.createStatement ();

// Call DefineColumnType to specify that the column will be
// retrieved as a String to avoid conversion from NUMBER to String
// on the client side. This also avoids a round-trip to the
// database to get the column type.
//
// There are 2 defineColumnType API. We use the one with 3 arguments.
// The 3rd argument allows us to specify the maximum length
// of the String. The values obtained for this column will
// not exceed this length.

((OracleStatement)stmt).defineColumnType (1, Types.VARCHAR, 7);

ResultSet rset = stmt.executeQuery ("select empno from emp");
while (rset.next ())
{
    System.out.println (rset.getString (1));
}

// Close the resultSet
rset.close();

// Close the statement
stmt.close ();

// Close the connection
conn.close();
}
}
```

Implicit Statement Caching—StmtCache1.java

This sample application uses implicit statement caching to create a result set from a database and then print out various information, including employee names.

For more information on implicit statement caching, see "[Using Implicit Statement Caching](#)" on page 13-8 and the other pertinent sections in [Chapter 13, "Statement Caching"](#).

```
/*
 * This sample to demonstrate Implicit Statement Caching. This can be
 * enabled by calling setStmtCacheSize on the Connection Object.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;
import oracle.jdbc.*;

class StmtCachel
{
    public static void main (String args [])
        throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        ((OracleConnection)conn).setStmtCacheSize(1);

        Connection sysconn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                                         "system", "manager");
        String sql = "select ENAME from EMP";

        System.out.println("Beging of 1st execution");
        getOpenCursors (sysconn);

        // Create a Statement
        PreparedStatement stmt = conn.prepareStatement (sql);
        System.out.println("1. Stmt is " + stmt);

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

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

        // Close the RseultSet
        rset.close();
    }
}
```

```
// Close the Statement
stmt.close();

System.out.println("End of 1st execution");
getOpenCursors (sysconn);

System.out.println("Reexecuting the same SQL");

stmt = conn.prepareStatement (sql);

System.out.println("2. Stmt is " + stmt);

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

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

// Close the ResultSet
rset.close();

// Close the Statement
stmt.close();

System.out.println("End of 2nd execution");
getOpenCursors (sysconn);

// Close the connection
conn.close();

System.out.println("After close of connection");
getOpenCursors (sysconn);

sysconn.close();
}

private static void getOpenCursors (Connection conn)
    throws SQLException
{
    System.out.println("Open Cusrors are : ");
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery ("select SQL_TEXT from V$OPEN_CURSOR");
    while (rs.next())
        System.out.println("Cursor's sql text is " + rs.getString(1));
}
```

```
        rs.close();
        rs = null;
        stmt.close();
        stmt = null;
    }
}
```

Explicit Statement Caching—StmtCache2.java

This sample application uses explicit statement caching to create a result set from a database and then print out various information including employee names.

For more information on implicit statement caching, see ["Using Explicit Statement Caching"](#) on page 13-9 and the other pertinent sections in [Chapter 13, "Statement Caching"](#).

```
/*
 * This sample to demonstrate Explicit Statement Caching. This can be
 * enabled by calling Oracle Specific calls like closeWithKey,
 * prepareStatementWithKey etc.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;
import oracle.jdbc.*;

class StmtCache2
{
    public static void main (String args [])
        throws SQLException
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        ((OracleConnection)conn).setStmtCacheSize(1);

        Connection sysconn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                                         "system", "manager");
        String sql = "select ENAME from EMP";
    }
}
```



```
System.out.println("Beging of 1st execution");
getOpenCursors (sysconn);

// Create a Statement
PreparedStatement stmt = conn.prepareStatement (sql);
System.out.println("1. Stmt is " + stmt);

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

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

// Close the RresultSet
rset.close();

// Close the Statement
//stmt.close();
((OracleStatement)stmt).closeWithKey ("mysql");

System.out.println("End of 1st execution");
getOpenCursors (sysconn);

System.out.println("Reexecuting the same SQL");

stmt = ((OracleConnection)conn).prepareStatementWithKey ("mysql");

System.out.println("2. Stmt is " + stmt);

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

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

// Close the RresultSet
rset.close();

// Close the Statement
stmt.close();

System.out.println("End of 2nd execution");
getOpenCursors (sysconn);
```

```
        // Close the connection
        conn.close();

        System.out.println("After close of connection");
        getOpenCursors (sysconn);

        sysconn.close();
    }

    private static void getOpenCursors (Connection conn)
        throws SQLException
    {
        System.out.println("Open Cusrors are : ");
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery ("select SQL_TEXT from V$OPEN_CURSOR");
        while (rs.next())
            System.out.println("Cursor's sql text is " + rs.getString(1));
        rs.close();
        rs = null;
        stmt.close();
        stmt = null;
    }
}
```

Samples for Connection Pooling and Distributed Transactions

This section includes samples of JDBC 2.0 extension features for data sources, connection pooling, connection caching, and distributed transactions (XA), as follows:

- [Data Source without JNDI—DataSource.java](#)
- [Data Source with JNDI—DataSourceJNDI.java](#)
- [Pooled Connection—PooledConnection.java](#)
- [Oracle Connection Cache \(dynamic\)—CCache1.java](#)
- [Oracle Connection Cache \("fixed with no wait"\)—CCache2.java](#)
- [XA with Suspend and Resume—XA2.java](#)
- [XA with Two-Phase Commit Operation—XA4.java](#)

Data Source without JNDI—DataSource.java

This example shows how to use JDBC 2.0 data sources without JNDI. For general information about data sources, including how to use them with or without JNDI, see "[Data Sources](#)" on page 14-2.

```
/**
 * A Simple DataSource sample without using JNDI.
 */

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

public class DataSource
{
    public static void main (String args [])
        throws SQLException
    {
        // Create a OracleDataSource instance explicitly
        OracleDataSource ods = new OracleDataSource();

        // Set the user name, password, driver type and network protocol
        ods.setUser("scott");
        ods.setPassword("tiger");
    }
}
```

```
ods.setDriverType("oci8");
ods.setNetworkProtocol("ipc");

// Retrieve a connection
Connection conn = ods.getConnection();
getUserName(conn);
// Close the connection
conn.close();
conn = null;
}

static void getUserName(Connection conn)
    throws SQLException
{
    // Create a Statement
    Statement stmt = conn.createStatement ();

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

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

    // Close the RresultSet
    rset.close();
    rset = null;

    // Close the Statement
    stmt.close();
    stmt = null;
}
}
```

Data Source with JNDI—DataSourceJNDI.java

This example shows how to use JDBC 2.0 data sources with JNDI. For general information about data sources, including how to use them with or without JNDI, see "[Data Sources](#)" on page 14-2.

This class includes `do_bind()` and `do_lookup()` methods for JNDI functionality, as well as a `getUserName()` method.

```
/**
```

```
* A Simple DataSource sample with JNDI.
* This is tested using File System based reference
* implementation of JNDI SPI driver from JavaSoft.
* You need to download fscontext1_2beta2.zip from
* JavaSoft site.
* Include providerutil.jar & fscontext.jar extracted
* from the above ZIP in the classpath.
* Create a directory /tmp/JNDI/jdbc
*/

// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;
import javax.naming.*;
import javax.naming.spi.*;
import java.util.Hashtable;

public class DataSourceJNDI
{
    public static void main (String args [])
        throws SQLException, NamingException
    {
        // Initialize the Context
        Context ctx = null;
        try {
            Hashtable env = new Hashtable (5);
            env.put (Context.INITIAL_CONTEXT_FACTORY,
                "com.sun.jndi.fscontext.RefFSContextFactory");
            env.put (Context.PROVIDER_URL, "file:/tmp/JNDI");
            ctx = new InitialContext(env);
        } catch (NamingException ne)
        {
            ne.printStackTrace();
        }

        do_bind(ctx, "jdbc/sampled");
        do_lookup(ctx, "jdbc/sampled");
    }

    static void do_bind (Context ctx, String ln)
        throws SQLException, NamingException
    {
```

```
// Create a OracleDataSource instance explicitly
OracleDataSource ods = new OracleDataSource();

// Set the user name, password, driver type and network protocol
ods.setUser("scott");
ods.setPassword("tiger");
ods.setDriverType("oci8");
ods.setNetworkProtocol("ipc");

// Bind it
System.out.println ("Doing a bind with the logical name : " + ln);
ctx.bind (ln,ods);
}

static void do_lookup (Context ctx, String ln)
    throws SQLException, NamingException
{
    System.out.println ("Doing a lookup with the logical name : " + ln);
    OracleDataSource ods = (OracleDataSource) ctx.lookup (ln);

    // Retrieve a connection
    Connection conn = ods.getConnection();
    getUsername(conn);
    // Close the connection
    conn.close();
    conn = null;
}

static void getUsername(Connection conn)
    throws SQLException
{
    // Create a Statement
    Statement stmt = conn.createStatement ();

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

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

    // Close the RresultSet
    rset.close();
    rset = null;
}
```

```
        // Close the Statement
        stmt.close();
        stmt = null;
    }
}
```

Pooled Connection—PooledConnection.java

This is a simple example of how to use JDBC 2.0 pooled connection functionality. For general information about connection pooling, see ["Connection Pooling"](#) on page 14-13.

```
/*
 * A simple Pooled Connection Sample
 */

import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;

class PooledConnection1
{
    public static void main (String args [])
        throws SQLException
    {

        // Create a OracleConnectionPoolDataSource instance
        OracleConnectionPoolDataSource ocpds =
            new OracleConnectionPoolDataSource();

        // Set connection parameters
        ocpds.setURL("jdbc:oracle:oci8:@");
        ocpds.setUser("scott");
        ocpds.setPassword("tiger");

        // Create a pooled connection
        PooledConnection pc = ocpds.getPooledConnection();

        // Get a Logical connection
        Connection conn = pc.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));

// Close the ResultSet
rset.close();
rset = null;

// Close the Statement
stmt.close();
stmt = null;

// Close the logical connection
conn.close();
conn = null;

// Close the pooled connection
pc.close();
pc = null;
}
}
```

OCI Connection Pool—OCIConnectionPool.java

This sample demonstrates OCI connection pooling.

For information on OCI connection pooling, see ["OCI Driver Connection Pooling"](#) on page 16-2.

```
/*
 * A simple OCI Connection Pool Sample
 */

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



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

        String url = "jdbc:oracle:oci:@";
        try {
            String url1 = System.getProperty("JDBC_URL");
            if (url1 != null)
                url = url1;
        } catch (Exception e) {
            // If there is any security exception, ignore it
            // and use the default
        }

        // Create an OracleOCIConnectionPool instance with default configuration
        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);

        // Get a connection from the pool
        OracleOCIConnection conn1 = (OracleOCIConnection)
        cpool.getConnection("scott", "tiger");

        // Create a Statement
        Statement stmt = conn1.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
        System.out.println ("-- Use the connection from the OracleOCIConnectionPool
        --");
        while (rset.next ())
            System.out.println (rset.getString (1));

        System.out.println ("-- Use another connection from the
        OracleOCIConnectionPool --");
    }
}
```

```
// Get another connection from the pool with different userID and password
OracleOCIConnection conn2 = (OracleOCIConnection)
cpool.getConnection("system", "manager");

// Create a Statement
stmt = conn2.createStatement ();

// Select the USER from DUAL to test the connection
rset = stmt.executeQuery ("select USER from DUAL");

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

// Reconfigure the OracleOCIConnectionPool in case the performance is too
bad.
// This might happen when many users are trying to connect at the same time.
// In this case, increase MAX_LIMIT to some larger number, and also increase
// INCREMENT to a positive number.

Properties p = new Properties();
p.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT,
Integer.toString(cpool.getMinLimit()));
p.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT,
Integer.toString(cpool.getMaxLimit() * 2)) ;
if (cpool.getConnectionIncrement() > 0)
    // Keep the old value
    p.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT,
Integer.toString(cpool.getConnectionIncrement()));
else
    // Set it to a number larger than 0
    p.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "1") ;

// Enable the new configuration
cpool.setPoolConfig(p);

// Print out the current configuration for the OracleOCIConnectionPool
System.out.println ("-- The new configuration for the
OracleOCIConnectionPool --");
displayPoolConfig(cpool);

// Close the ResultSet
rset.close();
rset = null;
```

```

// Close the Statement
stmt.close();
stmt = null;

// Close the connections
conn1.close();
conn2.close();
conn1 = null;
conn2 = null;

// Close the OracleOCIConnectionPool
cpool.close();
cpool = null;
}

// 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());
}
}

```

Middle-Tier Authentication—NtierAuth.java

This sample demonstrates middle-tier authentication through proxy connections. For information on middle-tier authentication, see ["Middle-Tier Authentication Through Proxy Connections"](#) on page 16-13.

```

/*
 * A Ntier Authentication Sample
 *
 */

import java.sql.*;
import javax.sql.*;

```

```

import java.util.Properties;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.oci.*;

class NtierAuth
{
    public static void main (String args [])
        throws SQLException
    {
        // Step 1: Connect as system/manager to create the users, setup roles and
        proxies.

        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

        String url = "jdbc:oracle:oci8:@";
        try {
            String url1 = System.getProperty("JDBC_URL");
            if (url1 != null)
                url = url1;
        } catch (Exception e) {
            // If there is any security exception, ignore it
            // and use the default
        }

        // Connect to the database as system/manager
        Connection sysConn = DriverManager.getConnection (url, "system", "manager");

        // Do some cleanup
        trySQL (sysConn, "drop role role1");
        trySQL (sysConn, "drop role role2");
        trySQL (sysConn, "drop user client cascade");
        trySQL (sysConn, "drop user proxy cascade");

        // Create a Statement
        Statement sysStmt = sysConn.createStatement ();

        // Create client and proxy
        sysStmt.execute("create user proxy identified by mehul");
        sysStmt.execute("create user client identified by ding");

        // Grant privilages to client and proxy
        sysStmt.execute("grant create session, connect, resource to proxy");
        sysStmt.execute("grant create session, connect, resource to client");
    }
}

```

```
// Create and setup roles with system connection
sysStmt.execute("create role role1");
sysStmt.execute("create role role2");

// Connect to the database as proxy
Connection proxyConn = DriverManager.getConnection (url, "proxy", "mehul");

// Create a table with proxy connection
Statement proxyStmt = proxyConn.createStatement ();
proxyStmt.execute("create table account (purchase number)");
proxyStmt.execute("insert into account values (6)");

// Grant privileges to role1, role2
proxyStmt.execute("grant select on account to role1");
proxyStmt.execute("grant insert on account to role2");

// Close the proxy statement and connection
proxyStmt.close();
proxyConn.close();

// Grant role1, role2 to client
sysStmt.execute("grant role1, role2 to client");

// Grant proxy privilege to connect as client
sysStmt.execute("alter user client grant connect through proxy with role
role1");

// Step 2: Use OCIConnectionPool to get the proxy connection

// Create an OracleOCIConnectionPool instance with default configuration
using proxy/mehul
OracleOCIConnectionPool cpool = new OracleOCIConnectionPool("proxy",
"mehul", url, null);

Properties prop = new Properties();
String[] roles = {"role1"};
prop.put(OracleOCIConnectionPool.PROXY_USER_NAME, "client" );
prop.put(OracleOCIConnectionPool.PROXY_ROLES, roles);

// Get the proxy connection
OracleOCIConnection conn = (OracleOCIConnection)
cpool.getProxyConnection(OracleOCIConnectionPool.PROXYTYPE_USER_NAME,
prop);
```

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

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

// Iterate through the result and print the purchase number
System.out.println ("-- Do a Select from the proxy connection --");
while (rset.next ())
    System.out.println (rset.getString (1));

// Close the RresultSet
rset.close();
rset = null;

// Now, try to do an Insert. This shouldn't be authorized
System.out.println ("-- Now, try to do an Insert with the proxy connection
--");
try {
    stmt.execute("insert into proxy.account values (9)");
} catch (SQLException e) {
    System.out.println ("Exception thrown: " + e.getMessage());
}
finally {
    if (stmt != null)
        stmt.close();
}

// Close the connection
conn.close();
conn = null;

// Close the OracleOCIConnectionPool
cpool.close();
cpool = null;

// Make the cleanup
trySQL (sysConn, "drop role role1");
trySQL (sysConn, "drop role role2");
trySQL (sysConn, "drop user client cascade");
trySQL (sysConn, "drop user proxy cascade");

// Close the system statement and connection
sysStmt.close();
sysConn.close();
```

```

    }

    // Used for Cleaning up the database
    private static void trySQL (Connection conn, String sqlString)
        throws SQLException
    {
        // Create a Statement
        Statement stmt = conn.createStatement ();

        try {
            stmt.execute(sqlString);
            stmt.close();
        } catch (SQLException e) {
            // In case the user or role hasn't been created, ignore it.
        }
        finally {
            if (stmt != null)
                stmt.close();
        }
    }
}

```

JDBC OCI Application Failover Callbacks—OCIFailOver.java

This sample demonstrates the registration and operation of JDBC OCI application failover callbacks.

For information on Transparent Application Failover (TAF) and failover events, see ["OCI Driver Transparent Application Failover"](#) on page 16-16.

```

/*
 * This sample demonstrates the registration and operation of
 * JDBC OCI application failover callbacks
 *
 * Note: Before you run this sample, set up the following
 *       service in tnsnames.ora:
 *       inst_primary=(DESCRIPTION=
 *           (ADDRESS=(PROTOCOL=tcp)(Host=hostname)(Port=1521))
 *           (CONNECT_DATA=(SERVICE_NAME=ORCL)
 *               (FAILOVER_MODE=(TYPE=SELECT)(METHOD=BASIC)))
 *       )
 *       )
 *
 * Please see the Oracle Net Administrator's Guide for more detail about

```

```

*          failover_mode
*
* To demonstrate the the functionality, first compile and start up the sample,
* then log into sqlplus and connect /as sysdba. While the sample is still
* running, shutdown the database with "shutdown abort;". At this moment,
* the failover callback functions should be invoked. Now, the database can
* be restarted, and the interrupted query will be continued.
*/

// You need to import java.sql and oracle.jdbc packages to use
// JDBC OCI failover callback

import java.sql.*;
import java.net.*;
import java.io.*;
import java.util.*;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleOCIFailover;

public class OCIFailOver {

    static final String user = "scott";
    static final String password = "tiger";
    static final String driver_class = "oracle.jdbc.OracleDriver";

    static final String URL = "jdbc:oracle:oci8:@inst_primary";

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

        Connection conn = null;
        Callback fcbk= new Callback();
        String msg = null;
        Statement stmt = null;
        ResultSet rset = null;

        // Load JDBC driver
        try {
            Class.forName(driver_class);
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}

```



```
// Connect to the database
conn = DriverManager.getConnection(URL, user, password);

// register TAF callback function
((OracleConnection) conn).registerTAFCallback(fcbk, msg);

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

for (int i=0; i<30; i++) {
    // Select the ENAME column from the EMP table
    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));

    // Sleep one second to make it possible to shutdown the DB.
    Thread.sleep(1000);
} // End for

// Close the RresultSet
rset.close();

// Close the Statement
stmt.close();

// Close the connection
conn.close();

} // End Main()

} // End class jdemofo

/*
 * Define class CallBack
 */
class CallBack implements OracleOCIFailover {

    // TAF callback function
    public int callbackFn (Connection conn, Object ctxt, int type, int event) {

        /*****

```

```

* There are 7 possible failover event
*   FO_BEGIN = 1   indicates that failover has detected a
*                   lost conenction and faiover is starting.
*   FO_END = 2     indicates successful completion of failover.
*   FO_ABORT = 3   indicates that failover was unsuccessful,
*                   and there is no option of retrying.
*   FO_REAUTH = 4  indicates that a user handle has been re-
*                   authenticated.
*   FO_ERROR = 5   indicates that failover was temporarily un-
*                   successful, but it gives the apps the opp-
*                   ortunity to handle the error and retry failover.
*                   The usual method of error handling is to issue
*                   sleep() and retry by returning the value FO_RETRY
*   FO_RETRY = 6
*   FO_EVENT_UNKNOWN = 7 It is a bad failover event
*****/
String failover_type = null;

switch (type) {
    case FO_SESSION:
        failover_type = "SESSION";
        break;
    case FO_SELECT:
        failover_type = "SELECT";
        break;
    default:
        failover_type = "NONE";
}

switch (event) {

    case FO_BEGIN:
        System.out.println(ctxt + ": "+ failover_type + " failing over...");
        break;
    case FO_END:
        System.out.println(ctxt + ": failover ended");
        break;
    case FO_ABORT:
        System.out.println(ctxt + ": failover aborted.");
        break;
    case FO_REAUTH:
        System.out.println(ctxt + ": failover.");
        break;
    case FO_ERROR:
        System.out.println(ctxt + ": failover error gotten. Sleeping...");

```

```

        // Sleep for a while
        try {
            Thread.sleep(100);
        }
        catch (InterruptedException e) {
            System.out.println("Thread.sleep has problem: " + e.toString());
        }
        return FO_RETRY;
    default:
        System.out.println(ctxt + ": bad failover event.");
        break;
    }

    return 0;
}
}

```

Oracle Connection Cache (dynamic)—CCache1.java

This is the first of two examples of connection caching using the Oracle sample implementation available with class `OracleConnectionCacheImpl`.

This example uses the dynamic scheme for situations where the maximum number of pooled connections has already been reached—new pooled connection instances are created as needed, but each one is automatically closed and freed as soon as the JDBC application is done using the logical connection instance that the pooled connection instance provided.

For information about connection caching in general and Oracle's sample implementation in particular, see "[Connection Caching](#)" on page 14-18.

```

/**
 * JDBC 2.0 Spec doesn't mandate that JDBC vendors implement a
 * Connection Cache. However, we implemented a basic one with two
 * schemes as an example.
 * A Sample demo to illustrate DYNAMIC_SCHEME of OracleConnectionCacheImpl.
 * Dynamic Scheme : This is the default scheme. New connections could be
 * created beyond the Max limit upon request but closed and freed when the
 * logical connections are closed. When all the connections are active and
 * busy, requests for new connections will end up creating new physical
 * connections. But these physical connections are closed when the
 * corresponding logical connections are closed. A typical grow and shrink

```

```

*  scheme.
*/

import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;

class CCache1
{
    public static void main (String args [])
        throws SQLException
    {
        OracleConnectionCacheImpl ods = new OracleConnectionCacheImpl();
        ods.setURL("jdbc:oracle:oci8:@");
        ods.setUser("scott");
        ods.setPassword("tiger");

        // Set the Max Limit
        ods.setMaxLimit (3);

        Connection conn1 = null;
        conn1 = ods.getConnection();
        if (conn1 != null)
            System.out.println("Connection 1 " + " Succeeded!");
        else
            System.out.println("Connection 1 " + " Failed !!!");

        Connection conn2 = null;
        conn2 = ods.getConnection();
        if (conn2 != null)
            System.out.println("Connection 2 " + " Succeeded!");
        else
            System.out.println("Connection 2 " + " Failed !!!");

        Connection conn3 = null;
        conn3 = ods.getConnection();
        if (conn3 != null)
            System.out.println("Connection 3 " + " Succeeded!");
        else
            System.out.println("Connection 3 " + " Failed !!!");

        Connection conn4 = null;
        conn4 = ods.getConnection();
        if (conn4 != null)

```

```

        System.out.println("Connection 4 " + " Succeeded!");
    else
        System.out.println("Connection 4 " + " Failed !!!");

    Connection conn5 = null;
    conn5 = ods.getConnection();
    if (conn5 != null)
        System.out.println("Connection 5 " + " Succeeded!");
    else
        System.out.println("Connection 5 " + " Failed !!!");

    System.out.println("Active size : " + ods.getActiveSize());
    System.out.println("Cache Size is " + ods.getCacheSize());

    // Close 3 logical Connections
    conn1.close();
    conn2.close();
    conn3.close();

    System.out.println("Active size : " + ods.getActiveSize());
    System.out.println("Cache Size is " + ods.getCacheSize());

    // close the Data Source
    ods.close();

    System.out.println("Active size : " + ods.getActiveSize());
    System.out.println("Cache Size is " + ods.getCacheSize());
}
}

```

Oracle Connection Cache ("fixed with no wait")—CCache2.java

This is the second of two examples of connection caching using the Oracle sample implementation available with class `OracleConnectionCacheImpl`.

This example uses the "fixed with no wait" scheme for situations where the maximum number of pooled connections has already been reached—a `null` is returned when a connection is requested.

For information about connection caching in general and Oracle's sample implementation in particular, see ["Connection Caching"](#) on page 14-18.

```
/**
```

```
* JDBC 2.0 Spec doesn't mandate that JDBC vendors implement a
* Connection Cache. However, we implemented a basic one with 2
* schemes as an Example.
* A Sample demo to illustrate FIXED_RETURN_NULL_SCHEME of
* OracleConnectionCacheImpl.
* Fixed with NoWait : At no instance there will be more active
* connections than the Maximum limit. Request for new connections
* beyond the max limit will return null.
*/

// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;

public class CCache2 {

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

        // Create a OracleConnectionPoolDataSource as an factory
        // of PooledConnections for the Cache to create.
        OracleConnectionPoolDataSource ocpds =
            new OracleConnectionPoolDataSource();
        ocpds.setURL("jdbc:oracle:oci8:@");
        ocpds.setUser("scott");
        ocpds.setPassword("tiger");

        // Associate it with the Cache
        OracleConnectionCacheImpl ods = new OracleConnectionCacheImpl(ocpds);

        // Set the Max Limit
        ods.setMaxLimit (3);

        // Set the Scheme
        ods.setCacheScheme (OracleConnectionCacheImpl.FIXED_RETURN_NULL_SCHEME);

        Connection conn = null;
        for (int i=0; i < 5; ++i )
        {
            conn = ods.getConnection();
            if (conn != null)
                System.out.println("Connection " + i + " Succeeded!");
        }
    }
}
```

```

        else
            System.out.println("Connection " + i + " Failed !!!");
        }

        System.out.println("Active size : " + ods.getActiveSize());
        System.out.println("Cache Size is " + ods.getCacheSize());

        // close the Data Source
        ods.close();

        System.out.println("Active size : " + ods.getActiveSize());
        System.out.println("Cache Size is " + ods.getCacheSize());
    }
}

```

XA with Suspend and Resume—XA2.java

This sample shows how to suspend and resume a transaction. It uses standard XA resource functionality to suspend and resume the transaction, but includes comments about how to use the Oracle extension `suspend()` and `resume()` methods as an alternative.

This class includes a `createXid()` method to form transaction IDs for purposes of this example.

For general information about distributed transactions and XA functionality, see [Chapter 15, "Distributed Transactions"](#).

```

/*
   A simple XA demo with suspend and resume. Opens 2 global
   transactions each of one branch. Does some DML on the first one
   and suspends it and does some DML on the 2nd one and resumes the
   first one and commits. Basically, to illustrate interleaving
   of global transactions.
   Need a java enabled 8.1.6 database to run this demo.
*/

// 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 XA2
{
    public static void main (String args [])
        throws SQLException
    {

        try
        {
            DriverManager.registerDriver(new OracleDriver());

            // You can put a database name after the @ sign in the connection URL.
            Connection conn =
                DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

            // Prepare a statement to create the table
            Statement stmt = conn.createStatement ();

            try
            {
                // Drop the test table
                stmt.execute ("drop table my_table");
            }
            catch (SQLException e)
            {
                // Ignore an error here
            }

            try
            {
                // Create a test table
                stmt.execute ("create table my_table (coll int)");
            }
            catch (SQLException e)
            {
                // Ignore an error here too
            }

            try
            {
                // Drop the test table
                stmt.execute ("drop table my_tab");
            }
        }
    }
}
```



```
}
catch (SQLException e)
{
    // Ignore an error here
}

try
{
    // Create a test table
    stmt.execute ("create table my_tab (coll int)");
}
catch (SQLException e)
{
    // Ignore an error here too
}

// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL("jdbc:oracle:oci8:@");
oxds.setUser("scott");
oxds.setPassword("tiger");

// get a XA connection
XAConnection pc = oxds.getXAConnection();
// Get a logical connection
Connection conn1 = pc.getConnection();

// Get XA resource handle
XAResource oxar = pc.getXAResource();
Xid xid1 = createXid(111,111);

// Start a transaction branch
oxar.start (xid1, XAResource.TMNOFLAGS);

// Create a Statement
Statement stmt1 = conn1.createStatement ();

// Do some DML
stmt1.executeUpdate ("insert into my_table values (2727)");

// Suspend the first global transaction
// ((OracleXAResource)oxar).suspend (xid1); or
oxar.end (xid1, XAResource.TMSUSPEND);

Xid xid2 = createXid(222,222);
```

```
oxar.start (xid2, XAResource.TMNOFLAGS);
Statement stmt2 = conn1.createStatement ();
stmt2.executeUpdate ("insert into my_tab values (7272)");
oxar.commit (xid2, true);
stmt2.close();
stmt2 = null;

// Close the Statement
stmt1.close();
stmt1 = null;

// Resume the first global transaction
// ((OracleXAResource)oxar).resume (xid1); or
oxar.start (xid1, XAResource.TMRESUME);

// End the branch
oxar.end(xid1, XAResource.TMSUCCESS);

// Do a 1 phase commit
oxar.commit (xid1, true);

// Close the connection
conn1.close();
conn1 = null;

// close the XA connection
pc.close();
pc = null;

ResultSet rset = stmt.executeQuery ("select col1 from my_table");
while (rset.next())
    System.out.println("Col1 is " + rset.getInt(1));

rset.close();
rset = null;

rset = stmt.executeQuery ("select col1 from my_tab");
while (rset.next())
    System.out.println("Col1 is " + rset.getString(1));

rset.close();
rset = null;

stmt.close();
stmt = null;
```

```

        conn.close();
        conn = 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());
        }
        xae.printStackTrace();
    }
}

static Xid createXid(int gd, int bd)
    throws XAException
{
    byte[] gid = new byte[1]; gid[0]= (byte) gd;
    byte[] bid = new byte[1]; bid[0]= (byte) bd;
    byte[] gtrid = new byte[64];
    byte[] bqual = new byte[64];
    System.arraycopy (gid, 0, gtrid, 0, 1);
    System.arraycopy (bid, 0, bqual, 0, 1);
    Xid xid = new OracleXid(0x1234, gtrid, bqual);
    return xid;
}
}

```

XA with Two-Phase Commit Operation—XA4.java

This example shows basic two-phase COMMIT functionality for a distributed transaction.

This class includes a `createXid()` method to form transaction IDs for purposes of this example. It also includes `doSomeWork1()` and `doSomeWork2()` methods to perform SQL operations.

For general information about distributed transactions and XA functionality, see [Chapter 15, "Distributed Transactions"](#).

```
/*
   A simple 2 phase XA demo. Both the branches talk to different RMS
   Need 2 java enabled 8.1.6 databases to run this demo.
   -> start-1
   -> start-2
   -> Do some DML on 1
   -> Do some DML on 2
   -> end 1
   -> end 2
   -> prepare-1
   -> prepare-2
   -> commit-1
   -> commit-2
   Please change the URL2 before running this.
*/

// 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:oci8:@";
            String URL2 =
                "jdbc:oracle:thin:@
                (description=(address=(host=dlsun991)(protocol=tcp)
                (port=5521))(connect_data=(sid=rdbms2)))";

            DriverManager.registerDriver(new OracleDriver());
```

```
// You can put a database name after the @ sign in the connection URL.
Connection conna =
    DriverManager.getConnection (URL1, "scott", "tiger");

// Prepare a statement to create the table
Statement stmta = conna.createStatement ();

Connection connb =
    DriverManager.getConnection (URL2, "scott", "tiger");

// 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 (coll 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 (coll char(30))");
    }
    catch (SQLException e)
    {
        // Ignore an error here too
    }

    // Create a XADataSource instance
    OracleXADataSource oxds1 = new OracleXADataSource();
    oxds1.setURL("jdbc:oracle:oci8:@");
    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 a XA connection to the underlying data source
    XAConnection pc1 = oxds1.getXAConnection();

    // We can use the same data source
    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);

    // Do something with conn1 and conn2

```

```
doSomeWork1 (conn1);
doSomeWork2 (conn2);

// END both the branches -- THIS IS MUST
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;

pcl.close();
pcl = null;
```

```

        pc2.close();
        pc2 = null;

        ResultSet rset = stmta.executeQuery ("select coll from my_table");
        while (rset.next())
            System.out.println("Coll is " + rset.getInt(1));

        rset.close();
        rset = null;

        rset = stmtb.executeQuery ("select coll from my_tab");
        while (rset.next())
            System.out.println("Coll 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
{
    byte[] gid = new byte[11]; gid[0]= (byte) 9;
}

```



```
byte[] bid = new byte[1]; bid[0]= (byte) bids;
byte[] gtrid = new byte[64];
byte[] bqual = new byte[64];
System.arraycopy (gid, 0, gtrid, 0, 1);
System.arraycopy (bid, 0, bqual, 0, 1);
Xid xid = new OracleXid(0x1234, gtrid, bqual);
return xid;
}

private static void doSomeWork1 (Connection conn)
    throws SQLException
{
    // Create a Statement
    Statement stmt = conn.createStatement ();

    int cnt = stmt.executeUpdate ("insert into my_table values (4321)");

    System.out.println("No of rows Affected " + cnt);

    stmt.close();
    stmt = null;
}

private static void doSomeWork2 (Connection conn)
    throws SQLException
{
    // Create a Statement
    Statement stmt = conn.createStatement ();

    int cnt = stmt.executeUpdate ("insert into my_tab values ('test')");

    System.out.println("No of rows Affected " + cnt);

    stmt.close();
    stmt = null;
}
}
```

HeteroRM XA—XA6.java

The following code example shows how to use the HeteroRM XA feature. All the XA-specific features that are available through JDBC XA are available in HeteroRM XA, such as commit, rollback, suspend, and resume.

For a complete description of HeteroRM XA, see ["OCI HeteroRM XA"](#) on page 16-19.

```
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 XA6
{
    public static void main (String args [])
        throws SQLException
    {

        try
        {
            DriverManager.registerDriver(new OracleDriver());

            String url = "jdbc:oracle:oci8:@";
            try {
                String url1 = System.getProperty("JDBC_URL");
                if (url1 != null)
                    url = url1;
            } catch (Exception e) {
                // If there is any security exception, ignore it
                // and use the default
            }

            // Connect to the database
            Connection conn =
                DriverManager.getConnection (url, "scott", "tiger");

            // Prepare a statement to create the table
            Statement stmt = conn.createStatement ();

            try
```

```
{
    // Drop the test table
    stmt.execute ("drop table my_table");
}
catch (SQLException e)
{
    // Ignore an error here
}

try
{
    // Create a test table
    stmt.execute ("create table my_table (coll int)");
}
catch (SQLException e)
{
    // Ignore an error here too
}

// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL(url);

// Set the nativeXA property to use HeteroRM XA feature
oxds.setNativeXA(true);

// Set the tnsEntry property to an older DB as required
oxds.setTNSEntryName("ora805");

oxds.setUser("scott");
oxds.setPassword("tiger");

// get a XA connection
XAConnection pc = oxds.getXAConnection();
// Get a logical connection
Connection conn1 = pc.getConnection();

// Get XA resource handle
XAResource oxar = pc.getXAResource();

Xid xid = createXid();

// Start a transaction branch
oxar.start (xid, XAResource.TMNOFLAGS);
```

```

        // Create a Statement
        Statement stmt1 = conn1.createStatement ();

        // Do some DML
        stmt1.executeUpdate ("insert into my_table values (7321)");

        // Close the Statement
        stmt1.close();
        stmt1 = null;

        // End the branch
        oxar.end(xid, XAResource.TMSUCCESS);

        // Do a 1 phase commit
        oxar.commit (xid, true);

        // Close the connection
        conn1.close();
        conn1 = null;

        // close the XA connection
        pc.close();
        pc = null;

        ResultSet rset = stmt.executeQuery ("select col1 from my_table");
        while (rset.next())
            System.out.println("Col1 is " + rset.getInt(1));

        rset.close();
        rset = null;

        stmt.close();
        stmt = null;

        conn.close();
        conn = null;

    } catch (SQLException sqe)                // check for SQLExceptions
    {
        sqe.printStackTrace();
    } catch (XAException xae)                // check for XAExceptions
    {
        xae.printStackTrace();
    }
}

```

```
static Xid createXid()
    throws XAException
{
    byte[] gid = new byte[1]; gid[0]= (byte) 1;
    byte[] bid = new byte[1]; bid[0]= (byte) 1;
    byte[] gtrid = new byte[64];
    byte[] bqual = new byte[64];
    System.arraycopy (gid, 0, gtrid, 0, 1);
    System.arraycopy (bid, 0, bqual, 0, 1);
    Xid xid = new OracleXid(0x1234, gtrid, bqual);
    return xid;
}
}
```

Sample Applet

This section demonstrates the use of the Oracle JDBC Thin driver for a simple applet that selects "Hello World" and the date from the database. Both the HTML page and applet code are shown here. A JDBC applet, like any typical applet, can be deployed using any standard Web server and run from any standard browser.

For a complete discussion of how to use JDBC with applets, see "[JDBC in Applets](#)" on page 18-16.

In this example, the Web server and database must be on the same host, as this is not a signed applet and does not use Oracle Connection Manager. For more information, see "[Connecting to a Database on a Different Host Than the Web Server](#)" on page 18-18.

HTML Page—JdbcApplet.htm

Here is the HTML code for the user interface for the applet.

```
<html>
<head>
<title>JDBC applet</title>
</head>
<body>

<h1>JDBC applet</h1>
```

This page contains an example of an applet that uses the Thin JDBC driver to connect to Oracle.<p>

The source code for the applet is in JdbcApplet.java. Please check carefully the driver class name and the connect string in the code.<p>

The Applet tag in this file contains a CODEBASE entry that must be set to point to a directory containing the Java classes from the Thin JDBC distribution *and* the compiled JdbcApplet.class.<p>

As distributed it will *not* work because the classes*.zip files are not in this directory.<p>

```
<hr>
<applet codebase="." archive="classes111.zip"
code="JdbcApplet" width=500 height=200>
</applet>
```

<hr>

Applet Code—JdbcApplet.java

Here is the source code for the applet.

```
/*
 * This sample applet just selects 'Hello World' and the date from the database
 */

// Import the JDBC classes
import java.sql.*;

// Import the java classes used in applets
import java.awt.*;
import java.io.*;
import java.util.*;

public class JdbcApplet extends java.applet.Applet
{

    // The connect string
    static final String connect_string =
        "jdbc:oracle:thin:scott/tiger@langer:5521:rdbsms";

    /* This is the kind of string you would use if going through the
     * Oracle connection manager which lets you run the database on a
     * different host than the Web Server. See the Oracle Net Administrator's
     * Guide for more information.
     * static final String connect_string = "jdbc:oracle:thin:scott/tiger@
     *      (description=(address_list=(address=(protocol=tcp)
     *      (host=dlsun511)(port=1610))(address=(protocol=tcp)
     *      (host=pkkrishna-pc2)(port=1521)))
     *      (source_route=yes)(connect_data=(sid=orcl)))";
     */

    // The query we will execute
    static final String query = "select 'Hello JDBC: ' || sysdate from dual";

    // The button to push for executing the query
    Button execute_button;

    // The place where to dump the query result
```

```
TextArea output;

// The connection to the database
Connection conn;

// Create the User Interface
public void init ()
{
    this.setLayout (new BorderLayout ());
    Panel p = new Panel ();
    p.setLayout (new FlowLayout (FlowLayout.LEFT));
    execute_button = new Button ("Hello JDBC");
    p.add (execute_button);
    this.add ("North", p);
    output = new TextArea (10, 60);
    this.add ("Center", output);
}

// Do the work
public boolean action (Event ev, Object arg)
{
    if (ev.target == execute_button)
    {
        try
        {

            // See if we need to open the connection to the database
            if (conn == null)
            {
                // Load the JDBC driver
                DriverManager.registerDriver (new oracle.jdbc.OracleDriver());

                // Connect to the database
                output.appendText ("Connecting to " + connect_string + "\n");
                conn = DriverManager.getConnection (connect_string);
                output.appendText ("Connected\n");
            }

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

            // Execute the query
            output.appendText ("Executing query " + query + "\n");
            ResultSet rset = stmt.executeQuery (query);
```



```
// Dump the result
while (rset.next ())
    output.appendText (rset.getString (1) + "\n");

// We're done
output.appendText ("done.\n");
}
catch (Exception e)
{
// Oops
output.appendText (e.getMessage () + "\n");
}
return true;
}
else
return false;
}
}
```

JDBC versus SQLJ Sample Code

This section contains a side-by-side comparison of two versions of the same sample code using Oracle `CustomDatum` functionality: one version is written in JDBC and the other in SQLJ. The objective of this section is to point out the differences in coding requirements between SQLJ and JDBC.

In the sample, two methods are defined: `getEmployeeAddress()` selects into a table and returns an employee's address based on the employee's number; `updateAddress()` takes the retrieved address, calls a stored procedure, and returns the updated address to the database.

In both versions of the sample code, these assumptions have been made:

- The `ObjectDemo.sql` SQL script (described below) has been run to create the necessary database entities.
- A PL/SQL stored function `UPDATE_ADDRESS`, which updates a given address, exists.
- The connection object (for JDBC) and default connection context (for SQLJ) have previously been created by the caller.
- Exceptions are handled by the caller.
- The value of the address argument (`addr`) passed to the `updateAddress` method can be null.

Note: The JDBC and SQLJ versions of the code are partial samples only. They cannot be run independently.

SQL Program to Create Tables and Objects

Following is a listing of the `ObjectDemo.sql` script that creates the tables and objects referenced by the two versions of the sample code. The `ObjectDemo.sql` script creates a `PERSON` object, an `ADDRESS` object, a typed table (`PERSONS`) of `PERSON` objects, and a relational table (`EMPLOYEES`) for employee data.

```
/** Using objects in SQLJ */
SET ECHO ON;
/**

/** Clean up */
DROP TABLE EMPLOYEES
/
```

```
DROP TABLE PERSONS
/
DROP TYPE PERSON FORCE
/
DROP TYPE ADDRESS FORCE
/

/** Create an address object */
CREATE TYPE address AS OBJECT
(
  street      VARCHAR(60),
  city        VARCHAR(30),
  state       CHAR(2),
  zip_code    CHAR(5)
)
/

/** Create a person object containing an embedded Address object */
CREATE TYPE person AS OBJECT
(
  name        VARCHAR(30),
  ssn         NUMBER,
  addr        address
)
/

/** Create a typed table for person objects */
CREATE TABLE persons OF person
/

/** Create a relational table with two columns that are REFs
    to person objects, as well as a column which is an Address object.*/

CREATE TABLE employees
( empnumber          INTEGER PRIMARY KEY,
  person_data        REF person,
  manager             REF person,
  office_addr        address,
  salary              NUMBER
)
/

/** insert code for UPDATE_ADDRESS stored procedure here
/
```

```

/**** Now let's put in some sample data
      Insert 2 objects into the persons typed table ****/

INSERT INTO persons VALUES (
    person('Wolfgang Amadeus Mozart', 123456,
    address('Am Berg 100', 'Salzburg', 'AU', '10424'))
/
INSERT INTO persons VALUES (
    person('Ludwig van Beethoven', 234567,
    address('Rheinallee', 'Bonn', 'DE', '69234'))
/

/** Put a row in the employees table **/

INSERT INTO employees (empnumber, office_addr, salary) " +
    " VALUES (1001, address('500 Oracle Parkway', " +
    " 'Redwood City', 'CA', '94065'), 50000)
/

/** Set the manager and person REFs for the employee **/

UPDATE employees
    SET manager =
        (SELECT REF(p) FROM persons p WHERE p.name = 'Wolfgang Amadeus Mozart')
/

UPDATE employees
    SET person_data =
        (SELECT REF(p) FROM persons p WHERE p.name = 'Ludwig van Beethoven')
/

COMMIT
/
QUIT

```

JDBC Version of the Sample Code

Following is the JDBC version of the sample code, which defines methods to retrieve an employee's address from the database, update the address, and return it to the database. Note, the "TO DOs" in the comment lines indicate where you might want to add additional code to enhance the usefulness of the code sample.

```

import java.sql.*;
import oracle.jdbc.*;

```

```
/**
 * This is what we have to do in JDBC
 */
public class SimpleDemoJDBC // line 7
{

//TO DO: make a main that calls this

public Address getEmployeeAddress(int empno, Connection conn)
    throws SQLException // line 13
{
    Address addr;
    PreparedStatement pstmt = // line 16
        conn.prepareStatement("SELECT office_addr FROM employees" +
            " WHERE empnumber = ?");
    pstmt.setInt(1, empno);
    OracleResultSet rs = (OracleResultSet)pstmt.executeQuery();
    rs.next(); // line 21
    //TO DO: what if false (result set contains no data)?
    addr = (Address)rs.getCustomDatum(1, Address.getFactory());
    //TO DO: what if additional rows?
    rs.close(); // line 25
    pstmt.close();
    return addr; // line 27
}

public Address updateAddress(Address addr, Connection conn)
    throws SQLException // line 30
{
    OracleCallableStatement cstmt = (OracleCallableStatement)
        conn.prepareCall("{ ? = call UPDATE_ADDRESS(?) }"); //line 34
    cstmt.registerOutParameter(1, Address._SQL_TYPECODE, Address._SQL_NAME);
    // line 36

    if (addr == null) {
        cstmt.setNull(2, Address._SQL_TYPECODE, Address._SQL_NAME);
    } else {
        cstmt.setCustomDatum(2, addr);
    }

    cstmt.executeUpdate(); // line 43
    addr = (Address)cstmt.getCustomDatum(1, Address.getFactory());
    cstmt.close(); // line 45
    return addr;
}
}
```

```
}  
}
```

Line 12: In the `getEmployeeAddress()` method definition, you must pass the connection object to the method definition explicitly.

Lines 16-20: Prepare a statement that selects an employee's address from the `EMPLOYEES` table on the basis of the employee number. The employee number is represented by a marker variable, which is set with the `setInt()` method. Note that because the prepared statement does not recognize the "INTO" syntax used in ["SQL Program to Create Tables and Objects"](#) on page 20-124, you must provide your own code to populate the address (`addr`) variable. Since the prepared statement is returning a custom object, cast the output to an Oracle result set.

Lines 21-23: Because the Oracle result set contains a custom object of type `Address`, use the `getCustomDatum()` method to retrieve it (the `Address` object could be created by `JPublisher`). The `getCustomDatum()` method requires you to use the static factory method `Address.getFactory()` to materialize an instance of an `Address` object. Since `getCustomDatum()` returns a `Datum`, cast the output to an `Address` object.

Note that the routine assumes a one-row result set. The "TO DOS" in the comment statements indicate that you must write additional code for the cases where the result set contains either no rows or more than one row.

Lines 25-27: Close the result set and prepared statement objects, then return the `addr` variable.

Line 29: In the `updateAddress()` definition, you must pass the connection object and the `Address` object explicitly.

The `updateAddress()` method passes an address to the database for update and fetches it back. The actual updating of the address is performed by the `UPDATE_ADDRESS` stored procedure (the code for this procedure is not illustrated in this example).

Line 33-43: Prepare an Oracle callable statement that takes an address object (`Address`) and passes it to the `UPDATE_ADDRESS` stored procedure. To register an object as an output parameter, you must know the object's SQL typecode and SQL type name.

Before passing the address object (`addr`) as an input parameter, the program must determine whether `addr` has a value or is null. Depending on the value of `addr`, the

program calls different `set` methods. If `addr` is null, the program calls `setNull()`; if it has a value, the program calls `setCustomDatum()`.

Line 44: Fetch the return result `addr`. Since the Oracle callable statement returns a custom object of type `Address`, use the `getCustomDatum()` method to retrieve it (the `Address` object could be created by `JPublisher`). The `getCustomDatum()` method requires you to use the static factory method `Address.getFactory` to materialize an instance of an `Address` object. Because `getCustomDatum()` returns a `Datum`, cast the output to an `Address` object.

Lines 45, 46: Close the Oracle callable statement, then return the `addr` variable.

Coding Requirements of the JDBC Version

Note the following coding requirements for the JDBC version of the sample code:

- The `getEmployeeAddress()` and `updateAddress()` definitions must explicitly include the connection object.
- Long SQL strings must be concatenated with the SQL concatenation character ("+").
- You must explicitly manage resources (for example, close result set and statement objects).
- You must cast datatypes as needed.
- You must know the `_SQL_TYPECODE` and `_SQL_NAME` of the factory objects that you are registering as output parameters.
- Null data must be explicitly handled.
- Host variables must be represented by parameter markers in callable and prepared statements.

Maintaining JDBC Programs

JDBC programs have the potential of being expensive in terms of maintenance. For example, in the above code sample, if you add another `WHERE` clause, then you must change the `SELECT` string. If you append another host variable, then you must increment the index of the other host variables by one. A simple change to one line in a JDBC program might require changes in several other areas of the program.

SQLJ Version of the Sample Code

Following is the SQLJ version of the sample code that defines methods to retrieve an employee's address from the database, update the address, and return it to the database.

```
import java.sql.*;

/**
 * This is what we have to do in SQLJ
 */
public class SimpleDemoSQLJ // line 6
{
    //TO DO: make a main that calls this?

    public Address getEmployeeAddress(int empno) // line 10
        throws SQLException
    {
        Address addr; // line 13
        #sql { SELECT office_addr INTO :addr FROM employees
              WHERE empnumber = :empno };
        return addr;
    } // line 18

    public Address updateAddress(Address addr)
        throws SQLException
    {
        #sql addr = { VALUES(UPDATE_ADDRESS(:addr)) }; // line 23
        return addr;
    }
}
```

Line 10: The `getEmployeeAddress()` method does not require a connection object. SQLJ uses a default connection context instance, which would have been defined previously somewhere in your application.

Lines 13-15: The `getEmployeeAddress()` method retrieves an employee address according to employee number. Use standard SQLJ `SELECT INTO` syntax to select an employee's address from the employee table if their employee number matches the one (`empno`) passed in to `getEmployeeAddress()`. This requires a declaration of the `Address` object (`addr`) that will receive the data. The `empno` and `addr` variables are used as input host variables. (Host variables are sometimes also referred to as bind variables.)

Line 16: The `getEmployeeAddress()` method returns the `addr` object.

Line 19: The `updateAddress()` method also uses the default connection context instance.

Lines 19-23: The address is passed to the `updateAddress()` method, which passes it to the database. The database updates it and passes it back. The actual updating of the address is performed by the `UPDATE_ADDRESS` stored function (the code for this function is not shown here). Use standard SQLJ function-call syntax to receive the address object (`addr`) output by `UPDATE_ADDRESS`.

Line 24: The `updateAddress()` method returns the `addr` object.

Coding Requirements of the SQLJ Version

Note the following coding requirements (and lack of requirements) for the SQLJ version of the sample code:

- An explicit connection is not required; a default connection context will have been defined previously in your application.
- No datatype casting is required.
- SQLJ does not require knowledge of `_SQL_TYPECODE`, `_SQL_NAME`, or factories.
- Null data is handled implicitly.
- No explicit code for resource management is required (for closing statements or result sets, for example).
- SQLJ embeds host variables, in contrast to JDBC, which uses parameter markers.
- String concatenation for long SQL statements is not required.
- You do not have to register output parameters.
- SQLJ syntax is simpler; for example, `SELECT INTO` syntax is supported and ODBC-style escapes are not used.

Reference Information

This chapter contains detailed JDBC reference information, including the following topics:

- [Valid SQL-JDBC Datatype Mappings](#)
- [Supported SQL and PL/SQL Datatypes](#)
- [Embedded SQL92 Syntax](#)
- [Oracle JDBC Notes and Limitations](#)
- [Related Information](#)

Valid SQL-JDBC Datatype Mappings

[Table 3–2](#) in [Chapter 3](#) describes the default mappings between Java classes and SQL datatypes supported by the Oracle JDBC drivers. Compare the contents of the **JDBC Datatypes**, **Standard Java Types** and **SQL Datatypes** columns in [Table 3–2](#) with the contents of [Table 21–1](#) below.

[Table 21–1](#) lists all the possible Java types to which a given SQL datatype can be validly mapped. The Oracle JDBC drivers will support these "non-default" 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 the following SQL datatypes, `oracle.sql.ORAData` or `oracle.sql.Datum` can be materialized as Java types.
 - For classes where `oracle.sql.ORAData` appears in *italics*, these can be generated by `JPublisher`.
-
-

Table 21–1 Valid SQL Datatype-Java Class Mappings

These SQL datatypes:	Can be materialized as these Java types:
CHAR, VARCHAR2, LONG	<code>oracle.sql.CHAR</code> <code>java.lang.String</code> <code>java.sql.Date</code> <code>java.sql.Time</code> <code>java.sql.Timestamp</code> <code>java.lang.Byte</code> <code>java.lang.Short</code> <code>java.lang.Integer</code> <code>java.lang.Long</code> <code>java.lang.Float</code> <code>java.lang.Double</code> <code>java.math.BigDecimal</code> <code>byte</code> , <code>short</code> , <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code>

Table 21–1 Valid SQL Datatype-Java Class Mappings (Cont.)

These SQL datatypes:	Can be materialized as these Java types:
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 (oracle.jdbc2.Blob under JDK 1.1.x)
CLOB	oracle.sql.CLOB java.sql.Clob (oracle.jdbc2.Clob under JDK 1.1.x)
Object types and SQLJ types	oracle.sql.STRUCT java.sql.Struct (oracle.jdbc2.Struct under JDK 1.1.x)

Table 21–1 Valid SQL Datatype-Java Class Mappings (Cont.)

These SQL datatypes:	Can be materialized as these Java types:
	java.sql.SqlData <i>oracle.sql.ORADATA</i>
Reference types	oracle.sql.REF java.sql.Ref (oracle.jdbc2.Ref under JDK 1.1.x) <i>oracle.sql.ORADATA</i>
Nested table types and VARRAY types	oracle.sql.ARRAY java.sql.Array (oracle.jdbc2.Array under JDK 1.1.x) <i>oracle.sql.ORADATA</i>

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.
- The mappings to `oracle.sql` classes are optimal if no conversion from SQL format to Java format is necessary.

Supported SQL and PL/SQL Datatypes

The tables in this section list SQL and PL/SQL datatypes, and whether the Oracle JDBC drivers and SQLJ support them. [Table 21–2](#) describes Oracle JDBC driver and SQLJ support for SQL datatypes.

Table 21–2 Support for SQL Datatypes

SQL Datatype	Supported by JDBC Drivers?	Supported by SQLJ?
BFILE	yes	yes
BLOB	yes	yes
CHAR	yes	yes
CLOB	yes	yes
DATE	yes	yes
NCHAR	no	no
NCHAR VARYING	no	no
NUMBER	yes	yes
NVARCHAR2	no	no
RAW	yes	yes
REF	yes	yes
ROWID	yes	yes
UROWID	no	no
VARCHAR2	yes	yes

[Table 21–3](#) describes Oracle JDBC driver and SQLJ support for the ANSI-supported SQL datatypes.

Table 21–3 Support for ANSI-92 SQL Datatypes

ANSI-Supported SQL Datatype	Supported by JDBC Drivers?	Supported by SQLJ?
CHARACTER	yes	yes
DEC	yes	yes
DECIMAL	yes	yes
DOUBLE PRECISION	yes	yes

Table 21–3 Support for ANSI-92 SQL Datatypes (Cont.)

ANSI-Supported SQL Datatype	Supported by JDBC Drivers?	Supported by SQLJ?
FLOAT	yes	yes
INT	yes	yes
INTEGER	yes	yes
NATIONAL CHARACTER	no	no
NATIONAL CHARACTER VARYING	no	no
NATIONAL CHAR	yes	yes
NATIONAL CHAR VARYING	no	no
NCHAR	yes	yes
NCHAR VARYING	no	no
NUMERIC	yes	yes
REAL	yes	yes
SMALLINT	yes	yes
VARCHAR	yes	yes

Table 21–4 describes Oracle JDBC driver and SQLJ support for SQL User-Defined types.

Table 21–4 Support for SQL User-Defined Types

SQL User-Defined type	Supported by JDBC Drivers?	Supported by SQLJ?
OPAQUE	yes	no
Reference types	yes	yes
SQLJ types (JAVA_STRUCT)	yes	no
Object types (JAVA_OBJECT)	yes	yes
Nested table types and VARRAY types	yes	yes

Note: SQLJ types are described in the *Information Technology - SQLJ - Part 2: SQL Types using the JAVATM Programming Language* document (ANSI NCITS 331.2-2000).

Table 21–5 describes Oracle JDBC driver and SQLJ support for PL/SQL datatypes. Note that PL/SQL datatypes include these categories:

- scalar types
- scalar character types (includes boolean and date datatypes)
- composite types
- reference types
- LOB types

Table 21–5 Support for PL/SQL Datatypes

PL/SQL Datatype	Supported by JDBC Drivers?	Supported by SQLJ?
Scalar Types:		
BINARY INTEGER	yes	yes
DEC	yes	yes
DECIMAL	yes	yes
DOUBLE PRECISION	yes	yes
FLOAT	yes	yes
INT	yes	yes
INTEGER	yes	yes
NATURAL	yes	yes
NATURAL _n	no	no
NUMBER	yes	yes
NUMERIC	yes	yes
PLS_INTEGER	yes	yes
POSITIVE	yes	yes
POSITIVE _n	no	no
REAL	yes	yes

Table 21–5 Support for PL/SQL Datatypes (Cont.)

PL/SQL Datatype	Supported by JDBC Drivers?	Supported by SQLJ?
SIGNTYPE	yes	yes
SMALLINT	yes	yes
Scalar Character Types:		
CHAR	yes	yes
CHARACTER	yes	yes
LONG	yes	yes
LONG RAW	yes	yes
NCHAR	no	no
NVARCHAR2	no	no
RAW	yes	yes
ROWID	yes	yes
STRING	yes	yes
UROWID	no	no
VARCHAR	yes	yes
VARCHAR2	yes	yes
BOOLEAN	yes	yes
DATE	yes	yes
Composite Types:		
RECORD	no	no
TABLE	no	no
VARRAY	yes	yes
Reference Types:		
REF CURSOR types	yes	yes
object reference types	yes	yes
LOB Types:		
BFILE	yes	yes
BLOB	yes	yes

Table 21–5 Support for PL/SQL Datatypes (Cont.)

PL/SQL Datatype	Supported by JDBC Drivers?	Supported by SQLJ?
CLOB	yes	yes
NCLOB	yes	yes

Notes:

- The types `NATURAL`, `NATURALn`, `POSITIVE`, `POSITIVEn`, and `SIGNTYPE` are subtypes of `BINARY_INTEGER`.
 - The types `DEC`, `DECIMAL`, `DOUBLE PRECISION`, `FLOAT`, `INT`, `INTEGER`, `NUMERIC`, `REAL`, and `SMALLINT` are subtypes of `NUMBER`.
-
-

Embedded SQL92 Syntax

Oracle's JDBC drivers support some embedded SQL92 syntax. This 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);
```

Note: Because prepared statements have usually been parsed prior to a call to `setEscapeProcessing()`, disabling escape processing for prepared statements will probably have no affect.

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".

This 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.
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:oci8:@", "scott", "tiger");

// 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-separated list of math functions supported by the driver. For example, `ABS(number)`, `COS(float)`, `SQRT(float)`.
- `getStringFunctions()`: Returns a comma-separated list of string functions supported by the driver. For example, `ASCII(string)`, `LOCATE(string1, string2, start)`.
- `getSystemFunctions()`: Returns a comma-separated list of system functions supported by the driver. For example, `DATABASE()`, `IFNULL(expression, value)`, `USER()`.
- `getTimeDateFunctions()`: Returns a comma-separated list of time and date functions supported by the driver. For example, `CURDATE()`, `DAYOFYEAR(date)`, `HOURL(time)`.

Oracle's JDBC drivers do not support the function keyword, 'fn'. If you try to use this keyword, for example:

```
{fn concat ("Oracle", "8i") }
```

Then you will get the error "Non supported SQL92 token at position xx: fn" when you run your Java application. The workaround is to use Oracle SQL syntax.

For example, instead of using the `fn` keyword in embedded SQL92 syntax:

```
Statement stmt = conn.createStatement ();
stmt.executeUpdate("UPDATE emp SET ename = {fn CONCAT('My', 'Name')}");
```

Use Oracle SQL syntax:

```
stmt.executeUpdate("UPDATE emp SET ename = CONCAT('My', 'Name')");
```

LIKE Escape Characters

The characters "%" and "_" have special meaning in SQL `LIKE` clauses (you use "%" to match zero or more characters, "_" to match exactly one character). If you want to interpret these characters literally in strings, you precede them with a special escape character. For example, if you want to use the ampersand "&" as the escape character, you identify it in the SQL statement as `{escape '&'}`:

```
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, you must enter it twice (that is, \\\). For example:

```
ResultSet rset = stmt.executeQuery("SELECT empno FROM emp
    WHERE ename LIKE '\\_%' {escape '\\'}");
```

Outer Joins

Oracle's JDBC drivers do not support outer join syntax: `{oj outer-join}`. 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's JDBC drivers support the following procedure and function call syntax:

Procedure calls (without a return value):

```
{ call procedure_name (argument1, argument2,...) }
```

Function calls (with a return value):

```
{ ? = 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 + " => " + new OracleSql().parse (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.

CursorName

Oracle JDBC drivers do not support the `getCursorName()` and `setCursorName()` methods, because there is no convenient way to map them to Oracle constructs. Oracle recommends using `ROWID` instead. For more information on how to use and manipulate `ROWIDs`, see ["Oracle ROWID Type"](#) on page 5-33.

SQL92 Outer Join Escapes

Oracle JDBC drivers do not support SQL92 outer join escapes. Use Oracle SQL syntax with `"(+)"` instead. For more information on SQL92 syntax, see ["Embedded SQL92 Syntax"](#) on page 21-10.

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. For a complete description of this, see ["Accessing PL/SQL Index-by Tables"](#) on page 16-21.

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 booleans, 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.

For an example of a workaround for `BOOLEAN`, see ["Boolean Parameters in PL/SQL Stored Procedures"](#) on page 19-9.

IEEE 754 Floating Point Compliance

The arithmetic for the Oracle `NUMBER` type does not comply with the IEEE 754 standard for floating-point arithmetic. Therefore, there can be small disagreements between the results of computations performed by Oracle and the same computations performed by Java.

Oracle stores numbers in a format compatible with decimal arithmetic and guarantees 38 decimal digits of precision. It represents zero, minus infinity, and plus infinity exactly. For each positive number it represents, it represents a negative number of the same absolute value.

It represents every positive number between 10^{-30} and $(1 - 10^{-38}) * 10^{126}$ to full 38-digit precision.

Catalog Arguments to DatabaseMetaData Calls

Certain `DatabaseMetaData` methods define a `catalog` parameter. This parameter is one of the selection criteria for the method. Oracle does not have multiple catalogs, but it does have packages. For more information on how the Oracle JDBC drivers treat the `catalog` argument, see "[DatabaseMetaData TABLE_REMARKS Reporting](#)" on page 12-26.

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.)

For information on how the Oracle JDBC drivers handle errors, see "[Processing SQL Exceptions](#)" on page 3-33.

Bind by Name

Binding by name is not supported. Under certain circumstances, previous versions of the Oracle JDBC drivers have allowed binding statement variables by name. 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 is not part of the JDBC specification, either 1.0 or 2.0, and Oracle does not support it. The JDBC drivers can throw a `SQLException` or produce unexpected results.

Prior releases of the Oracle JDBC drivers did not retain bound values from one call of `execute` to the next as specified in JDBC 1.0. Bound values are now retained. 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 this release, 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 `execute` calls will send `NULL` as the value of the argument.

Related Information

This section lists Web sites that contain useful information for JDBC programmers. Many of the sites are referenced in other sections of this manual. In this list you can find references to the Oracle JDBC drivers, Oracle SQLJ, Java technology, the Java Developer's Kit APIs (for versions 1.2.x and 1.1.x), the Java Security API, and resources to help you write signed applets.

Oracle JDBC Drivers and SQLJ

Oracle JDBC Driver Home Page (Oracle Corporation)

<http://www.oracle.com/java/jdbc>

Oracle SQLJ Home Page (Oracle Corporation)

<http://www.oracle.com/java/sqlj>

Java Technology

Java Technology Home Page (Sun Microsystems, Inc.):

<http://www.javasoft.com>

Java Development Kit (JDK1.2.x and 1.1.x) (Sun Microsystems, Inc.):

<http://java.sun.com/products/jdk>

A

Row Set

This appendix describes the following topics:

- [Row Set Setup and Configuration](#)
- [Runtime Properties for Row Set](#)
- [Row Set Listener](#)
- [Traversing Through the Rows](#)
- [Cached Row Set](#)
- [JDBC Row Set](#)

Introduction

A *row set* is an object which encapsulates a set of rows. These rows are accessible through the `javax.sql.RowSet` interface. This interface supports component models of development, like JavaBeans, and is part of JDBC optional package by JavaSoft.

Three kinds of row set are supported by JavaSoft:

- Cached row set
- JDBC row set
- Web row set

Note: Oracle implements cached row set and JDBC row set, but *not* Web row set.

All the row set implementation is in the `oracle.jdbc.rowset` package. *Web row set* is a semi-connected row set. It has a servlet which has a connection open and the `WebRowSet` interface translates the user calls to appropriate request to the servlet over HTTP. This is targeted at Thin clients which have only HTTP implementation in them.

Note: The row set feature is available only in JDK 1.2 or later.

The `RowSet` interface provides a set of properties which can be altered to access the data in the database through a single interface. It supports properties and events which forms the core of JavaBeans. It has various properties like connect string, user name, password, type of connection, the query string itself, and also the parameters passed to the query. The following code executes a simple query:

```
...
rowset.setUrl ("jdbc:oracle:oci8:@");
rowset.setUsername ("SCOTT");
rowset.setPassword ("TIGER");
rowset.setCommand (
    "SELECT empno, ename, sal FROM emp WHERE empno = ?");

// empno of employee name "KING"
rowset.setInt (1, 7839);
...
```


In the above example, the URL, user name, password, SQL query, and bind parameter required for the query are set as the command properties to retrieve the employee name and salary. Also, the row set would contain `empno`, `ename`, and `sal` for the employee with the `empno` as 7839 and whose name is KING.

Row Set Setup and Configuration

The classes for the row set feature are found in a separate archive, `ocrs12.zip`. This file is located in the `$ORACLE_HOME/jdbc` directory. To use row set, you need to include this archive in your `CLASSPATH`.

For Unix (sh), the command is:

```
CLASSPATH=$CLASSPATH:$ORACLE_HOME/jdbc/lib/ocrs12.zip
export CLASSPATH
```

For Windows 2000/NT/98/95, the command is:

```
set CLASSPATH=%CLASSPATH%;%ORACLE_HOME%\jdbc\lib\ocrs12.zip
```

This might also be set in the project properties in case you are using an IDE like Jdeveloper.

Oracle row set interfaces are implemented in the `oracle.jdbc.rowset` package. Import this package to use any of the Oracle row set implementations.

The `OracleCachedRowSet` and `OracleJDBCRowSet` classes implement the `javax.sql.RowSet` interface, which extends `java.sql.ResultSet`. Row set not only provides the interfaces of result set, but also some of the properties of the `java.sql.Connection` and `java.sql.PreparedStatement` interfaces. Connections and prepared statements are totally abstracted by this interface. `CachedRowSet` is serializable. This class implements the `java.io.Serializable` interface. This enables the `OracleCachedRowSet` class to be moved over the network or other JVM sessions.

Also available is the `oracle.jdbc.rowset.OracleRowSetListenerAdapter` class.

Runtime Properties for Row Set

Typically, static properties for the applications can be set for a row set at the development time and the rest of the properties which are dynamic (are dependent on runtime) can be set at the runtime. The static properties may include the connection URL, username, password, connection type, concurrency type of the row set, or the query itself. The runtime properties, like the bind parameters for the query, could be bound at runtime. Scenarios where the query itself is a dynamic property is also common.

Row Set Listener

The row set feature supports multiple listeners to be registered with the `RowSet` object. Listeners can be registered using the `addRowSetListener()` method and unregistered through the `removeRowSetListener()` method. A listener should implement the `javax.sql.RowSetListener` interface to register itself as the row set listener. Three types of events are supported by the `RowSet` interface:

1. `cursorMoved` event : Generated whenever there is a cursor movement, which occurs when the `next()` or `previous()` methods are called
2. `rowChanged` event : Generated when a new row is inserted, updated, or deleted from the row set
3. `rowsetChanged` event : Generated when the whole row set is created or changed

The following code shows the registration of a row set listener:

```
MyRowSetListener rowsetListener =
    new MyRowSetListener ();
// adding a rowset listener.
rowset.addRowSetListener (rowsetListener);

// implementation of a rowset listener
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 which handle only a few events can implement only the required events by using the `OracleRowSetAdapter` 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. The remaining events are not handled by the application.

```
rowset.addRowSetListener (new OracleRowSetAdapter ()
{
    public void rowSetChanged(RowSetEvent event)
    {
        // your action for rowsetChanged
    }
});
```

Traversing Through the Rows

Various methods to traverse through the rows are provided by the `RowSet` interface. These properties are inherited directly from the `java.sql.ResultSet` interface. The `RowSet` interface could be used as a `ResultSet` interface for retrieval and updating of data. The `RowSet` interface provides an optional way to implement a scrolling and updatable result set if they are not provided by the result set implementation.

Note: The scrollable properties of the `java.sql.ResultSet` interface are also provided by the Oracle implementation of `ResultSet`.

Cached Row Set

A *cached row set* is a row set implementation where the rows are cached and the row set does not have a live connection to the database (disconnected) and it is a serializable object, which provides the standard interface as of the `javax.sql.RowSet` interface. `OracleCachedRowSet` is the implementation of `CachedRowSet` by Oracle, and the `OracleCachedRowSet` is used interchangeably with `CachedRowSet`.

In the following code, an `OracleCachedRowSet` object is created and the connection URL, username, password, and the SQL query for the row set is set as properties. The `RowSet` object is populated through the `execute` method. After the `execute` call, 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:oci8:@");
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 connection `Url`, `Username`, `Password`, `connection type` (optional), and the query string as properties for the `RowSet` object.
3. Invoke the `execute()` method to populate the `RowSet` object.

`CachedRowSet` can be populated with the existing `ResultSet` object, using the `populate()` method, as shown in the following code:

```
// 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 above example, a `ResultSet` object is obtained by executing a query and the retrieved `ResultSet` object is passed to the `populate()` method of the cached row set to populate the contents of the result set into cached row set.

To populate a `CachedRowSet` object with an already available result set, complete the following steps:

1. Instantiate `OracleCachedRowSet`.
2. Pass the already available `ResultSet` object to the `populate()` method to populate the `RowSet` object.

All the interfaces provided by the `ResultSet` interface are implemented in `RowSet`. The following code shows how to scroll through a row set:

```
/**
 * Scrolling forward, and printing the empno in
 * the order in which it was fetched.
 */
// going to the first row of the rowset
rowset.beforeFirst ();
while (rowset.next ())
    System.out.println ("empno: " +rowset.getInt (1));
```

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, since the connection or result set on which the property applies would have already been created.

In the example above, the cursor position is initialized to the position before the first row of the row set by the `beforeFirst()` method. The rows are retrieved in forward direction using the `next()` method.

```
/**
 * 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 above example, the cursor position is initialized to the position after the last row of the RowSet. The rows are retrieved in reverse direction using the `previous()` method of RowSet.

Inserting, updating, and deleting rows are supported by the row set feature as they are in the result set feature. The following code illustrates the insertion of a row at the fifth position of a row set:

```

/**
 * 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 above example, a call to the `absolute()` method with a parameter 5 takes the cursor to the fifth position of the row set and a call to the `moveToInsertRow()` method creates a place for the insertion of a new row into the row set. The `updateXXX()` 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 row set. The changes are committed through `acceptChanges()` method.

The following code shows 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 above example, a `FileOutputStream` object is opened for a `emp_tab.dmp` file, and the populated `OracleCachedRowSet` object is written to the file using `ObjectOutputStream`. This is retrieved using `FileInputStream` and the `ObjectInputStream` objects.

`OracleCachedRowSet` takes care of the serialization of non-serializable form of data like `InputStream`, `OutputStream`, `BLOBS` and `CLOBS`.

`OracleCachedRowSets` also implements meta data of its own, which could be obtained without any extra server roundtrip. The following code shows how you can obtain meta data for the row set:

```

ResultSetMetaData metaData = rowset.getMetaData ();
int maxCol = metaData.getColumnCount ();
for (int i = 1; i <= maxCol; ++i)
    System.out.println ("Column (" + i + ") "
        + metaData.getColumnName (i));

```

The above example illustrates how to retrieve a `ResultSetMetaData` object and print the column names in the `RowSet`.

Since the `OracleCachedRowSet` class is serializable, it can be passed across a network or between JVMs, as done in Remote Method Invocation (RMI). Once the `OracleCachedRowSet` class is populated, it can move around any JVM, or any environment which does not have JDBC drivers. Committing the data in the row set (through the `acceptChanges ()` method) 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 row set 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 row set without any connection to the database. Whenever data is committed to the database, the `acceptChanges()` method is called which synchronizes the data in the row set 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) or a Network Computer (NC).

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 cached row set are the following:

- Cloning a row set
- Creating a copy of a row set
- Creating a shared copy of a row set

CachedRowSet Constraints

All the constraints which apply to updatable result set are applicable here, except serialization, since `OracleCachedRowSet` is serializable. The SQL query has the following constraints:

- References only a single table in the database
- Contain no join operations
- Selects the primary key of the table it references

In addition, a SQL query should also satisfy the conditions below if inserts are to be performed:

- Selects all of the 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 since all the data is cached in memory.

Properties which apply to the connection cannot be set after populating the row set since the properties cannot be applied to the connection after retrieving the data from the same like, transaction isolation and concurrency mode of the result set.

JDBC Row Set

A *JDBC row set* is another row set implementation. It is a simple, non-serializable connected row set which provides JDBC interfaces in the form of a Bean interface. Any call to `JDBCRowSet` percolates directly to the JDBC interface. The usage of the JDBC interface is the same as any other row set implementation.

[Table A-1](#) shows how the `JDBCRowSet` interface differs from `CachedRowSet` interface.

Table A-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	No	Yes	No	No	Yes
Cached	Yes	No	Yes	Yes	No

The JDBC row set is a connected row set which has a live connection to the database and all the calls on the JDBC row set are percolated to the mapping call in JDBC connection, statement, or result set. A cached row set does not have any connection to the database open.

JDBC row set requires the presence of JDBC drivers where a cached row set does not require JDBC drivers during manipulation, but during population of the row set and the committing the changes of the row set.

The following code shows how a JDBC row set is used:

```
RowSet rowset = new OracleJDBCRowSet ();
rowset.setUrl ("java:oracle:oci8:@");
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 above example, the connection URL, username, password, and the SQL query is set as the connection properties to the row set and the query is executed through the `execute()` method and the rows are retrieved and printed.

JDBC Error Messages

This appendix briefly discusses the general structure of 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](#)
- [TTC Messages](#)

Each of the two message lists is first sorted by ORA number, and then alphabetically.

For general information about processing JDBC exceptions, see "[Processing SQL Exceptions](#)" on page 3-33.

Note: "Cause" and "Action" information for each message will be provided in a later release.

General Structure of JDBC Error Messages

The general JDBC error message structure allows runtime 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 ORA number, and then alphabetically.

JDBC Messages Sorted by ORA Number

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-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 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
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-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 Caching is not enabled for this Connection object
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

JDBC Messages Sorted Alphabetically

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-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-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-17087	Ignored performance hint: setFetchDirection()
ORA-17049	Inconsistent java and sql object types
ORA-17072	Inserted value too large for column
ORA-17001	Internal Error
ORA-17089	internal error
ORA-17045	Internal error: Attempt to access bind values beyond the batch value
ORA-17044	Internal error: Data array not allocated
ORA-17046	Internal error: Invalid index for data access
ORA-17058	Internal error: Invalid NLS Conversion ratio
ORA-17068	Invalid argument(s) in call
ORA-17080	invalid batch command
ORA-17042	Invalid Batch Value
ORA-17055	Invalid character encountered in
ORA-17003	Invalid column index
ORA-17006	Invalid column name
ORA-17004	Invalid column type
ORA-17106	invalid combination of connections 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-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-17043	Invalid stream maximum size
ORA-17064	Invalid Sytnax or Database name is null
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-17050	no such element in vector
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-17096	Statement Caching cannot be enabled for this logical connection
ORA-17095	Statement Caching is not enabled for this Connection object
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-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

HeteroRM XA Messages

The following are the JDBC error messages that are specific to the HeteroRM XA feature.

HeteroRM XA Messages Sorted by ORA Number

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_INVALID 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_INVALID during xa_close
ORA-17236	C-XA returned XAER_PROTO during xa_close

HeteroRM XA Messages Sorted Alphabetically

ORA-17203	Could not cast pointer type to jlong
ORA-17235	C-XA returned XAER_INVALID during xa_close
ORA-17215	C-XA returned XAER_INVALID during xa_open

ORA-17236	C-XA returned XAER_PROTO during xa_close
ORA-17216	C-XA returned XAER_PROTO during xa_open
ORA-17233	C-XA returned XAER_RMERR during xa_close
ORA-17213	C-XA returned XAER_RMERR during xa_open
ORA-17206	Failed to obtain OCIEnv handle from C-XA using xaoEnv
ORA-17205	Failed to obtain OCISvcCtx handle from C-XA using xaoSvcCtx
ORA-17204	Input array too short to hold OCI handles
ORA-17207	The tnsEntry property was not set in DataSource
ORA-17202	Unable to properly convert RM name from Java to C
ORA-17201	Unable to properly convert XA close string from Java to C
ORA-17200	Unable to properly convert XA open string from Java to C

TTC Messages

This section lists TTC error messages, first sorted by ORA number, and then alphabetically.

TTC Messages Sorted by ORA Number

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-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 Alphabetically

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	Exceeding key size
ORA-17406	Exceeding maximum buffer length
ORA-17416	FATAL
ORA-17414	Insufficient Buffer size to store Columns Names
ORA-17438	Internal - Unexpected value
ORA-17418	Internal structure's field length error
ORA-17433	invalid arguments in call
ORA-17409	invalid buffer length
ORA-17422	Invalid class in factory
ORA-17419	Invalid number of columns returned
ORA-17435	invalid number of in_out_binds in IOV
ORA-17436	invalid number of outbinds
ORA-17432	invalid options in all7
ORA-17439	Invalid SQL type
ORA-17408	invalid Type Representation(getRep)
ORA-17407	invalid Type Representation(setRep)
ORA-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

Index

Symbols

\$, 14-9
%, 8-55, 21-13
&, 21-13
+, 21-16
-, 21-13

A

absolute positioning in result sets, 11-2
absolute() method (result set), 11-14
acceptChanges() method, A-13
ACID properties, 17-2
addBatch() method, 12-11
addConnectionEventListener() method (connection cache), 14-22
addRowSetListener() method, A-6
afterLast() method (result sets), 11-14
ANO (Oracle Advanced Security), 18-9
ANSI Web site, xxvii, 8-53
APPLET HTML tag, 18-25
applets
 connecting to a database, 18-16
 deploying in an HTML page, 18-25
 packaging, 18-24
 for JDK 1.2.x or 1.1.x browser, 18-24
 packaging and deploying, 1-13
 signed applets
 browser security, 18-21
 object-signing certificate, 18-21
 using signed applets, 18-20
 using with firewalls, 18-21
 working with, 18-16

ARCHIVE, parameter for APPLET tag, 18-26
ARRAY
 class, 5-12
 descriptors, 5-12
 objects, creating, 5-12, 10-12
array descriptor
 creating, 10-22
ArrayDescriptor object, 10-11, 10-22
 creating, 10-12
 deserialization, 10-15
 get methods, 10-14
 serialization, 10-15
 setConnection() method, 10-15
arrays
 defined, 10-2
 example program, 20-35
 getting, 10-20
 named, 10-2
 passing to callable statement, 10-23
 retrieving from a result set, 10-16
 retrieving partial arrays, 10-19
 using type maps, 10-25
 working with, 10-2
ASO (Oracle Advanced Security), 18-9
authentication (security), 18-10
AUTHENTICATION_LEVEL parameter, 18-19
auto-commit mode
 disabling, 19-6
 result set behavior, 19-6

B

batch updates--see update batching
batch value

- checking value, 12-7
- connection batch value, setting, 12-5
- connection vs. statement value, 12-4
- default value, 12-5
- overriding value, 12-7
- statement batch value, setting, 12-6
- BatchUpdateException, 12-16
- beforeFirst() method, A-10
- beforeFirst() method (result sets), 11-13
- begin method, 17-3, 17-16, 17-19
- BFILE
 - accessing data, 7-25
 - class, 5-12
 - creating and populating columns, 7-23
 - defined, 3-28
 - example program, 20-40
 - introduction, 7-2
 - locators, 7-20
 - getting from a result set, 7-20
 - getting from callable statement, 7-21
 - passing to callable statements, 7-21
 - passing to prepared statements, 7-21
 - manipulating data, 7-25
 - reading data, 7-22
- BFILE locator, selecting, 5-13
- BigDecimal mapping (for attributes), 8-47
- bindds command, 14-9, 17-14, 17-25
 - example, 14-11
 - options, 14-10
- bindut command, 17-12, 17-17, 17-24
- BLOB, 7-5
 - class, 5-12
 - creating and populating, 7-10
 - creating columns, 7-11
 - getting locators, 7-3
 - introduction, 7-2
 - locators
 - getting from result set, 7-4
 - selecting, 5-13
 - manipulating data, 7-12
 - populating columns, 7-11
 - reading data, 7-6, 7-8
 - writing data, 7-9
- Boolean parameters, restrictions, 19-9
- branch qualifier (distributed transactions), 15-13

C

- cache schemes (connection cache), 14-27
- CachedRowSet, A-9
- caching, client-side
 - custom use for scrollable result sets, 11-6
 - Oracle use for scrollable result sets, 11-5
- callable statement
 - getting a BFILE locator, 7-21
 - getting LOB locators, 7-4
 - passing BFILE locator, 7-21
 - passing LOB locators, 7-5
 - using getOracleObject() method, 6-5
- cancelRowUpdates() method (result set), 11-20
- casting return values, 6-10
- catalog arguments (DatabaseMetaData), 21-17
- CHAR class
 - conversions with KPRB driver, 18-34
- CHAR columns
 - NLS size restrictions, Thin, 18-6
 - space padding, 19-8
 - using setFixedCHAR() to match in WHERE, 6-17
- character sets, 5-32
 - conversions with KPRB driver, 18-34
- checksums
 - code example, 18-14
 - setting parameters in Java, 18-14
 - support by OCI drivers, 18-12
 - support by Thin driver, 18-13
- Class.forName method, 3-3
- CLASSPATH, specifying, 2-6
- clearBatch() method, 12-14
- clearDefines() method, 12-24
- clearMetaData parameter, 13-6, 16-11
- client installation, 1-12
- CLOB
 - class, 5-12
 - creating and populating, 7-10
 - creating columns, 7-11
 - introduction, 7-2
 - locators, 7-3
 - getting from result set, 7-4
 - passing to callable statements, 7-5
 - passing to prepared statement, 7-5

- locators, selecting, 5-13
- manipulating data, 7-12
- populating columns, 7-11
- reading data, 7-6, 7-9
- writing data, 7-9
- close(), 13-5
- close() method, 5-19, 5-20, 5-21, 19-8
 - for caching statements, 13-7, 13-9
 - for OracleConnectionCache interface, 14-24
- closeFile() method, 7-26
- closePooledConnection() method, 14-24
- closeWithKey(), 13-5
- closeWithKey() method, 13-10
- CMAN.ORA file, creating, 18-19
- CODE, parameter for APPLET tag, 18-25
- CODEBASE, parameter for APPLET tag, 18-25
- collections
 - defined, 10-2
- collections (nested tables and arrays), 10-11
- column types
 - defining, 12-23
 - redefining, 12-20
- commit a distributed transaction branch, 15-12
- commit changes to database, 3-13
- commit method, 17-3, 17-4, 17-16, 17-19
- Common Object Request Broker Architecture (CORBA), 1-17
- CONCUR_READ_ONLY result sets, 11-9
- CONCUR_UPDATABLE result sets, 11-9
- concurrency types in result sets, 11-4
- connect string
 - for KPRB driver, 18-29
 - for the Oracle8 Connection Manager, 18-19
- connection
 - closing, 3-14
 - from KPRB driver, 1-15
 - opening, 3-3
 - opening for JDBC OCI driver, 3-9
 - opening for JDBC Thin driver, 3-10
 - Properties object, 3-6
- connection caching
 - adding connection event listener, 14-22
 - basics, accessing the cache, 14-19
 - basics, closing connections, 14-20
 - basics, opening connections, 14-19
 - basics, setting up a cache, 14-18
 - cache instance getConnection() method, 14-19
 - connection events, 14-20
 - creating connection event listener, 14-22
 - implementation scenarios, 14-20
 - OracleConnectionCache interface, 14-24
 - OracleConnectionCacheImpl class, 14-25
 - OracleConnectionEventListener class, 14-29
 - overview, 14-18
 - preliminary steps, 14-21
 - removing connection event listener, 14-23
 - steps in closing a connection, 14-23
 - steps in opening a connection, 14-21
- connection event listener, 14-22
- Connection Manager, 1-12, 18-17, 18-18
 - installing, 18-18
 - starting, 18-19
 - using multiple managers, 18-20
 - writing the connect string, 18-19
- connection methods, JDBC 2.0 result sets, 11-32
- connection pooling
 - concepts, 14-13
 - creating data source and connecting, 14-16
 - introduction, 14-13
 - Oracle data source implementation, 14-14
 - pooled connections, 14-15
 - sample application, 20-89
 - standard data source interface, 14-14
- connection properties
 - database, 3-7
 - defaultBatchValue, 3-7
 - defaultRowPrefetch, 3-7
 - includeSynonyms, 3-7
 - internal_logon, 3-7
 - sysdba, 3-7
 - sysoper, 3-7
 - password, 3-6
 - put() method, 3-9
 - remarksReporting, 3-7
 - user, 3-6
- connectionClosed() method (connection event listener), 14-29
- connectionErrorOccurred() method (connection event listener), 14-29
- connections

- read-only, 19-14
- constants for SQL types, 5-22
- CORBA, 1-17
- CREATE DIRECTORY statement
 - for BFILEs, 7-23
- CREATE TABLE statement
 - to create BFILE columns, 7-23
 - to create BLOB, CLOB columns, 7-11
- CREATE TYPE command, 8-53, 8-55, 8-63
- CREATE TYPE statement, 8-29, 8-52
- create() method
 - for ORADDataFactory interface, 8-21
- createDescriptor() method, 8-5, 8-61, 10-14
- createStatement(), 13-5
- createStatement() method, 5-18, 13-10
- createStatementWithKey(), 13-5
- createStatementWithKey() method, 13-10, 13-11
- createTemporary() method, 7-18
- creationState() method, 13-7
 - code example, 13-7
- CursorName
 - limitations, 21-16
- cursors, 19-8
- custom collection classes
 - and JPublisher, 10-27
 - defined, 10-2, 10-27
- custom Java classes, 5-4
 - creating, 20-43, 20-47
 - defined, 8-2
- custom object classes
 - creating, 8-10
 - defined, 8-2
- custom reference classes
 - and JPublisher, 9-10
 - defined, 9-2, 9-10

D

- data conversions, 6-2
 - LONG, 3-20
 - LONG RAW, 3-20
- data sources
 - creating and connecting (with JNDI), 14-8
 - creating and connecting (without JNDI), 14-7
 - logging and tracing, 14-11

- Oracle implementation, 14-3
- PrintWriter, 14-11
- properties, 14-4
- sample application (with JNDI), 20-85
- sample application (without JNDI), 20-86
- standard interface, 14-3
- data streaming
 - avoiding, 3-24
 - example program, 20-18
- database
 - connecting
 - from an applet, 18-16
 - via multiple Connection Managers, 18-20
 - with server-side internal driver, 18-27
 - connection testing, 2-8
- database connection
 - connection property, 3-7
- database meta data methods, JDBC 2.0 result sets, 11-35
- database URL
 - including userid and password, 3-5
- database URL, specifying, 3-5
- DatabaseMetaData calls, 21-17
- DatabaseMetaData class, 21-12
 - entry points for applets, 18-24
- DataSource object
 - binding in namespace, 17-14, 17-18
 - create dynamically, 17-25
 - getConnection method, 17-6
- datatype classes, 5-7
- datatype mappings, 3-16
- datatypes
 - Java, 3-16
 - Java native, 3-16
 - JDBC, 3-16
 - Oracle SQL, 3-16
- DATE class, 5-13
- DBMS_LOB package, 7-6
- debugging JDBC programs, 19-11
- DEFAULT_CHARSET character set value, 5-32
- defaultBatchValue connection property, 3-7
- defaultConnection() method, 18-27
- defaultRowPrefetch connection property, 3-7
- defineColumnType() method, 3-24, 5-19, 12-24
- DELETE in a result set, 11-18

- deleteRow() method (result set), 11-18
- deletesAreDetected() method (database meta data), 11-29
- deserialization
 - ArrayDescriptor object, 10-15
 - creating a StructDescriptor object, 8-6
 - creating an ArrayDescriptor object, 10-15
 - definition of, 8-6, 10-15
 - StructDescriptor object, 8-6
- distributed transaction ID component, 15-13
- distributed transactions
 - branch qualifier, 15-13
 - check for same resource manager, 15-13
 - commit a transaction branch, 15-12
 - components and scenarios, 15-2
 - concepts, 15-3
 - distributed transaction ID component, 15-13
 - end a transaction branch, 15-10
 - example of implementation, 15-18
 - global transaction identifier, 15-13
 - ID format identifier, 15-13
 - introduction, 15-2
 - Oracle XA connection implementation, 15-7
 - Oracle XA data source implementation, 15-6
 - Oracle XA ID implementation, 15-13
 - Oracle XA optimizations, 15-17
 - Oracle XA resource implementation, 15-8
 - prepare a transaction branch, 15-11
 - roll back a transaction branch, 15-12
 - sample application (suspend/resume), 20-105
 - sample application (two-phase commit), 20-1, 20-109
 - start a transaction branch, 15-10
 - transaction branch ID component, 15-13
 - XA connection interface, 15-7
 - XA data source interface, 15-6
 - XA error handling, 15-16
 - XA exception classes, 15-15
 - XA ID interface, 15-13
 - XA resource functionality, 15-9
 - XA resource interface, 15-8
- DriverManager class, 3-3
 - getConnection method, 17-6
- driverType, 14-5
- dynamic SQL, 1-2

- DYNAMIC_SCHEME (connection cache), 14-28

E

- EJB, A-13
- encryption
 - code example, 18-14
 - overview, 18-11
 - setting parameters in Java, 18-14
 - support by OCI drivers, 18-12
 - support by Thin driver, 18-13
- end a distributed transaction branch, 15-10
- Enterprise Java Beans (EJB), A-13
- environment variables
 - specifying, 2-6
- errors
 - general JDBC message structure, B-2
 - general JDBC messages, listed, B-3
 - processing exceptions, 3-33
 - TTC messages, listed, B-14
- exceptions
 - printing stack trace, 3-34
 - retrieving error code, 3-33
 - retrieving message, 3-33
 - retrieving SQL state, 3-33
- execute() method, A-16
- executeBatch() method, 12-13
- executeQuery() method, 5-19
- executeUpdate() method, 12-9
- explicit statement caching, 13-2, 20-82
 - definition of, 13-4
 - null data, 13-11
- extensions to JDBC, Oracle, 5-1, 6-1, 8-1, 9-1, 10-1, 12-1
- external changes (result set)
 - defined, 11-27
 - seeing, 11-28
 - visibility vs. detection, 11-29
- external file
 - defined, 3-28
- EXTERNAL NAME clause, 8-55

F

- fetch direction in result sets, 11-17

- fetch size, result sets, 11-24
- finalizer methods, 19-8
- firewalls
 - configuring for applets, 18-22
 - connect string, 18-23
 - described, 18-21
 - required rule list items, 18-22
 - using with applets, 1-13, 18-21
- first() method (result sets), 11-14
- FIXED_RETURN_NULL_SCHEME (connection cache), 14-28
- FIXED_WAIT_SCHEME (connection cache), 14-28
- floating-point compliance, 21-17
- format identifier, transaction ID, 15-13
- forward-only result sets, 11-3
- freeTemporary() method, 7-18
- function call syntax, SQL92 syntax, 21-14

G

- getActiveSize() method (connection cache), 14-28
- getARRAY() method, 10-16
- getArray() method, 10-6, 10-10, 10-16
 - using type maps, 10-18
- getArrayType() method, 10-14
- getAsciiOutputStream() method, 7-15
 - for writing CLOB data, 7-7
- getAsciiStream() method, 7-15
 - for reading CLOB data, 7-7
- getAttributes() method, 8-3
 - used by Structs, 8-15
- getAutoBuffering() method
 - of the oracle.sql.ARRAY class, 10-9
 - of the oracle.sql.STRUCT class, 8-9
- getBaseName() method, 10-14
- getBaseType() method, 10-6, 10-14, 10-20
- getBaseTypeName() method, 9-4, 10-6
- getBinaryOutputStream() method, 7-14
 - for writing BLOB data, 7-7
- getBinaryStream() method, 3-22, 7-14, 7-26
 - for reading BFILE data, 7-22
 - for reading BLOB data, 7-6
- getBufferSize() method, 7-14, 7-15
- getBytes() method, 3-23, 5-10, 7-14, 7-26
- getCacheSize() method (connection cache), 14-28
- getCharacterOutputStream() method, 7-15
 - for writing CLOB data, 7-7
- getCharacterStream() method, 7-15
 - for reading CLOB data, 7-7
- getChars() method, 7-15
- getChunkSize() method, 7-14, 7-16
- getColumnCount() method, 6-19
- getColumnName() method, 6-19
- getColumns() method, 12-26
- getColumnType() method, 6-19
- getColumnTypeName() method, 6-19
- getConcurrency() method (result set), 11-12
- getConnection() method, 3-4, 10-15, 16-11, 18-27
- getCursor() method, 5-35, 5-36
- getCursorName() method
 - limitations, 21-16
- getDefaultExecuteBatch() method, 5-18, 12-7
- getDefaultRowPrefetch() method, 5-18, 12-21
- getDescriptor() method, 8-4, 10-6
- getDirAlias() method, 7-25, 7-27
- getErrorCode() method (SQLException), 3-33
- getExecuteBatch() method, 5-20, 12-6, 12-7
- getFetchSize() method, 11-24
- getJavaSqlConnection() method, 8-4, 10-6
- getJavaSqlConnection() method, 5-25
- getLanguage() method, 8-62
- getMaxLength() method, 10-14
- getMessage() method (SQLException), 3-33
- getMetaData() method, 8-62
- getName() method, 7-25, 7-27
- getNumericFunctions() method, 21-12
- getObject() method
 - casting return values, 6-10
 - for object references, 9-6
 - for ORADData objects, 8-22
 - for SQLInput streams, 8-16
 - for SQLOutput streams, 8-17
 - for Struct objects, 8-7
 - return types, 6-4, 6-6
 - to get BFILE locators, 7-20
 - to get Oracle objects, 8-8
 - used with ORADData interface, 8-24
- getOracleArray() method, 10-6, 10-16, 10-19
- getOracleAttributes() method, 8-4, 8-8
- getOracleObject() method, 5-20, 5-22

- casting return values, 6-10
- return types, 6-4, 6-6
- using in callable statement, 6-5
- using in result set, 6-5
- getOraclePlsqlIndexTable() method, 16-22, 16-25, 16-26
 - argument
 - int paramIndex, 16-26
 - code example, 16-27
- getORADData() method, 8-22, 8-24
- getPassword() method, 14-5
- getPlsqlIndexTable() method, 16-22, 16-25, 16-27
 - arguments
 - Class primitiveType, 16-28
 - int paramIndex, 16-27
 - code example, 16-26, 16-28
- getProcedureColumns() method, 12-26
- getProcedures() method, 12-26
- getREF() method, 9-7
- getRemarksReporting() method, 5-19
- getResultSet() method, 5-19, 10-6
- getRow() method (result set), 11-15
- getRowPrefetch() method, 5-19, 12-21
- getSQLState() method (SQLException), 3-33
- getSQLTypeName() method, 8-3, 10-6, 10-20
- getStatus method, 17-5
- getStmtCacheSize() method
 - code example, 13-7
- getString() method, 5-32
 - to get ROWIDs, 5-33
- getStringFunctions() method, 21-12
- getStringWithReplacement() method, 5-32
- getSTRUCT() method, 8-7
- getSubString() method, 7-16
 - for reading CLOB data, 7-7
- getSystemFunctions() method, 21-12
- getTimeDateFunctions() method, 21-12
- getTransactionIsolation() method, 5-18, 19-14
- getType() method (result set), 11-12
- getTypeMap() method, 5-18, 8-13
- getUpdateCounts() method
 - (BatchUpdateException), 12-17
- getValue() method, 9-5
 - for object references, 9-6
- getXXX() methods

- casting return values, 6-10
 - for specific datatypes, 6-7
 - Oracle extended properties, 14-6
- global transaction identifier (distributed transactions), 15-13
- global transactions, 15-2

H

- HEIGHT, parameter for APPLET tag, 18-25
- HeteroRM XA, 16-19
- HeuristicMixedException, 17-4
- HeuristicRollbackException, 17-4
- HTML tags, to deploy applets, 18-25
- HTTP, 1-17
- http
 - //www.ansi.org/, 8-53
- HTTP protocol, 1-7
- Hypertext Transfer Protocol (HTTP), 1-17

I

- IDE, A-4
- IEEE 754 floating-point compliance, 21-17
- IIOp, 1-17
- IllegalStateException, 17-4
- implicit statement caching, 13-2
 - code example, 20-79
 - concept graphic, 13-3
 - definition of, 13-2
 - Least Recently Used (LRU) scheme, 13-4
- IN OUT parameter mode, 16-24
 - code example, 20-11
- IN parameter mode, 16-22
 - code example, 20-11
- includeSynonyms connection property, 3-7
- INSERT in a result set, 11-21
- INSERT INTO statement
 - for creating BFILE columns, 7-24
- insertRow() method (result set), 11-22
- insertsAreDetected() method (database meta data), 11-29
- installation
 - client, 1-12
 - directories and files, 2-5

- verifying on the client, 2-5
- integrity
 - code example, 18-14
 - overview, 18-11
 - setting parameters in Java, 18-14
 - support by OCI drivers, 18-12
 - support by Thin driver, 18-13
- internal changes (result set)
 - defined, 11-27
 - seeing, 11-27
- internal_logon connection property, 3-7
 - sysdba, 3-7
 - sysoper, 3-7
- Internet Inter-ORB Protocol (IIOP), 1-17
- isAfterLast() method (result set), 11-15
- isBeforeFirst() method (result set), 11-15
- isFileOpen() method, 7-27
- isFirst() method (result set), 11-15
- isLast() method (result set), 11-15
- isSameRM() (distributed transactions), 15-13
- isTemporary() method, 7-18

J

- Java
 - compiling and running, 2-7
 - datatypes, 3-16
 - native datatypes, 3-16
 - stored procedures, 3-32
 - stream data, 3-19
- Java Naming and Directory Interface (JNDI), 14-2
- Java Sockets, 1-7
- Java virtual machine (JVM), 1-10, 18-27
- JavaBeans, A-2
- java.math, Java math packages, 3-2
- JavaSoft, A-2
- java.sql, JDBC packages, 3-2
- java.sql.SQLData, 8-53
- java.sql.SQLException() method, 3-33
- java.sql.Types class, 12-24
- java.util.Dictionary class
 - used by type maps, 8-12
- java.util.Map class, 10-19
- java.util.Properties, 16-7
- JDBC

- and IDEs, 1-17
- and Oracle Application Server, 1-17
- basic program, 3-2
- datatypes, 3-16
- defined, 1-2
- guidelines for using, 1-4
- importing packages, 3-2
- limitations of Oracle extensions, 21-16
- sample files, 2-7
- testing, 2-8
- JDBC 2.0 support
 - datatype support, 4-3
 - extended feature support, 4-5
 - introduction, 4-2
 - JDK 1.2.x vs. JDK 1.1.x, 4-3
 - overview of features, 4-7
 - standard feature support, 4-4
- JDBC drivers
 - and NLS, 18-3
 - applets, 1-12
 - applications, 1-12
 - choosing a driver for your needs, 1-10
 - common features, 1-6
 - common problems, 19-8
 - compatibilities, 2-2
 - determining driver version, 2-8
 - introduction, 1-5
 - registering, 3-3
 - requirements, 2-2
 - restrictions, 19-9
 - SQL92 syntax, 21-10
- JDBC mapping (for attributes), 8-46
- JdbcCheckup program, 2-8
- JDBCSpy, 19-13
- JDBCTest, 19-13
- JDeveloper, 1-17
- Jdeveloper, A-4
- JDK
 - migration from 1.1.x to 1.2.x, 4-5
 - versions supported, 1-16
- JNDI
 - looking up data source, 14-11
 - overview of Oracle support, 14-2
 - registering data source, 14-9
- JPublisher, 5-4, 8-25, 8-45

- JPublisher utility, 5-4, 8-10
 - creating custom collection classes, 10-27
 - creating custom Java classes, 8-45
 - creating custom reference classes, 9-10
 - SQL type categories and mapping options, 8-46
 - type mapping modes and settings, 8-46
 - type mappings, 8-45

JTA

- client-side demarcation, 17-3, 17-15
- enlisting resources, 17-5
- limitations, 17-8
- nested transactions, 17-8
- overview, 17-2
- specification web site, 17-1
- two-phase commit, 17-21
 - overview, 17-7

- JVM, 1-10, 18-27

K

KPRB driver

- connection string for, 18-29
- described, 1-10, 18-27
- NLS considerations, 18-4
- relation to the SQL engine, 18-27
- session context, 18-31
- testing, 18-31
- transaction context, 18-31

L

- last() method (result set), 11-14
- LD_LIBRARY_PATH variable, specifying, 2-7
- Least Recently Used (LRU) scheme, 13-4, 16-8
- length() method, 7-14, 7-16, 7-27, 10-6
- libheteroxa9_g.so Solaris shared library, 16-19
- libheteroxa9.so Solaris shared library, 16-19
- LIKE escape characters, SQL92 syntax, 21-13
- limitations on setBytes() and setString(), use of streams to avoid, 3-30
- loadjava tool, 8-55
- LOB
 - defined, 3-27
 - introduction, 7-2
 - locators, 7-2

- reading data, 7-6

LOB locators

- getting from callable statements, 7-4
- passing, 7-5

LOBs

- empty, 7-17
- example program, 20-25

locators

- getting for BFILES, 7-20
- getting for BLOBs, 7-3
- getting for CLOBs, 7-3
- LOB, 7-2
- passing to callable statements, 7-5
- passing to prepared statement, 7-5

- logging with a data source, 14-11

- logical connection instance, 14-13

LONG

- data conversions, 3-20

LONG RAW

- data conversions, 3-20

- LRU scheme, 13-4, 16-8

M

- make() method, 5-31

- memory leaks, 19-8

- migration from JDK 1.1.x to 1.2.x, 4-5

- moveToCurrentRow() method (result set), 11-21

- moveToInsertRow() method (result set), 11-21

- mutable arrays, 10-27

N

- named arrays, 10-2

- defined, 10-11

- Native Method Interface, 1-16

- nativeXA, 14-6, 16-19

- NC, A-13

- Network Computer (NC), A-13

- network events, trapping, 19-11

- next() method, A-10

- next() method (result set), 11-15

NLS

- and JDBC drivers, 18-3

- conversions, 18-3

- for JDBC OCI drivers, 18-3
 - for JDBC Thin drivers, 18-4
 - for KPRB driver, 18-4
- Java methods that employ, 18-2
- Thin driver CHAR/VARCHAR2 size restrictions, 18-6
- using, 18-2
- NLS_LANG environment variable, 18-3
- NMI (Native Method Interface), 1-16
- NotSupportedException, 17-4
- NULL data
 - converting, 6-2
- null data
 - explicit statement caching, 13-11
- NUMBER class, 5-13

O

- OAS, 1-17
- object references
 - accessing object values, 9-7, 9-9
 - described, 9-2
 - passing to prepared statements, 9-8
 - retrieving, 9-6
 - retrieving from callable statement, 9-7
 - updating object values, 9-7, 9-9
- object-JDBC mapping (for attributes), 8-46
- OCI driver
 - applications, 1-12
 - described, 1-8
 - NLS considerations, 18-3
- ODBCSpy, 19-13
- ODBCTest, 19-13
- openFile() method, 7-26
- optimization, performance, 19-6
- Oracle Advanced Security, 1-12
 - support by JDBC, 18-9
 - support by OCI drivers, 18-9
 - support by Thin driver, 18-10
- Oracle Application Server, 1-17
- Oracle Application Server (OAS), 1-17
- Oracle datatypes
 - using, 6-1
- Oracle extensions
 - datatype support, 5-3
 - limitations, 21-16
 - catalog arguments to DatabaseMetaData calls, 21-17
 - CursorName, 21-16
 - IEEE 754 floating-point compliance, 21-17
 - PL/SQL TABLE, BOOLEAN, RECORD types, 21-16
 - read-only connection, 19-14
 - SQL92 outer join escapes, 21-16
 - SQLWarning class, 21-17
 - object support, 5-4
 - packages, 5-2
 - result sets, 6-3
 - schema naming support, 5-5
 - statements, 6-3
 - support under 8.0.x/7.3.x drivers, 5-36
 - to JDBC, 5-1, 6-1, 8-1, 9-1, 10-1, 12-1
- Oracle mapping (for attributes), 8-46
- Oracle Net
 - name-value pair, 3-4
 - protocol, 1-7
- Oracle objects
 - and JDBC, 8-2
 - converting with ORaData interface, 8-21
 - converting with SQLData interface, 8-15
 - getting with getObject() method, 8-8
 - Java classes which support, 8-3
 - mapping to custom object classes, 8-10
 - reading data by using SQLData interface, 8-17
 - working with, 8-2
 - writing data by using SQLData interface, 8-20
- Oracle SQL datatypes, 3-16
- Oracle8 Connection Manager, 18-17
- OracleCallableStatement interface, 5-20
 - getOraclePlsqlIndexTable() method, 16-22
 - getPlsqlIndexTable() method, 16-22
 - getTimeStamp(), 5-14
 - getTimeStampTZ(), 5-14
 - getTimeStampTZ(), 5-14
 - getXXX() methods, 6-7
 - registerIndexTableOutParameter() method, 16-22, 16-24
 - registerOutParameter() method, 6-13
 - setPlsqlIndexTable() method, 16-22
- OracleCallableStatement object, 13-3, 13-4

- OracleConnection class, 5-18
- OracleConnection interface, 16-4
- OracleConnection object, 13-2
- OracleConnectionCache interface, 14-24
 - close() method, 14-24
 - closePooledConnection() method, 14-24
 - reusePooledConnection() method, 14-24
- OracleConnectionCacheImpl class, 14-25, 14-27
 - getActiveSize() method, 14-28
 - getCacheSize() method, 14-28
 - instantiating and setting properties, 14-25
 - schemes for new pooled connections, 14-27
 - setCacheScheme() method, 14-28
 - setConnectionPoolDataSource() method, 14-26
 - setMaxLimit() method
 - setMaxLimit() method (connection cache), 14-27
 - setMinLimit() method
 - setMinLimit() method (connection cache), 14-27
 - setting maximum pooled connections, 14-26
 - setting minimum pooled connections, 14-27
- OracleConnectionCacheImpl interface, 16-4
- OracleConnectionEventListener
 - connectionClosed() method, 14-29
- OracleConnectionEventListener class, 14-29
 - connectionErrorOccurred() method, 14-29
 - instantiating, 14-29
 - setDataSource() method, 14-29
- OracleConnectionPoolDataSource class, 14-14
- OracleDatabaseMetaData class, 21-12
 - and applets, 18-24
- OracleDataSource class, 14-3, 16-4
- OracleDriver class, 5-17
 - defaultConnection method, 17-6
- oracle.jdbc. package, 5-16
- oracle.jdbc., Oracle JDBC extensions, 3-3
- oracle.jdbc2 package, described, 5-26
- oracle.jdbc2.Struct class, 5-10
 - getAttributes() method, 8-3
 - getSQLTypeName() method, 8-3
- oracle.jdbc.OracleCallableStatement interface, 5-20
 - close() method, 5-21
 - getOracleObject() method, 5-20
 - getXXX() methods, 5-21, 5-22
 - registerOutParameter() method, 5-21
 - setNull() method, 5-21
 - setOracleObject() methods, 5-21
 - setXXX() methods, 5-21
- oracle.jdbc.OracleConnection interface, 5-18
 - createStatement() method, 5-18
 - getDefaultExecuteBatch() method, 5-18
 - getDefaultRowPrefetch() method, 5-18
 - getRemarksReporting() method, 5-19
 - getTransactionIsolation() method, 5-18, 19-14
 - getTypeMap() method, 5-18
 - prepareCall() method, 5-18
 - prepareStatement() method, 5-18
 - setDefaultExecuteBatch() method, 5-18
 - setDefaultRowPrefetch() method, 5-19
 - setRemarksReporting() method, 5-19
 - setTransactionIsolation() method, 5-18, 19-14
 - setTypeMap() method, 5-18
- oracle.jdbc.OracleDriver class, 5-17
- oracle.jdbc.OraclePreparedStatement interface, 5-19
 - close() method, 5-20
 - getExecuteBatch() method, 5-20
 - setExecuteBatch() method, 5-20
 - setNull() method, 5-20
 - setOracleObject() method, 5-20
 - setORADData() method, 5-20
 - setXXX() methods, 5-20
- oracle.jdbc.OracleResultSet interface, 5-21
 - getOracleObject() method, 5-22
- oracle.jdbc.OracleResultSetMetaData interface, 5-22, 6-19
 - getColumnCount() method, 6-19
 - getColumnName() method, 6-19
 - getColumnType() method, 6-19
 - getColumnTypeName() method, 6-19
 - using, 6-19
- oracle.jdbc.OracleSql class, 21-14
- oracle.jdbc.OracleStatement interface, 5-19
 - close() method, 5-19
 - defineColumnType(), 5-19
 - executeQuery() method, 5-19
 - getResultSet() method, 5-19
 - getRowPrefetch() method, 5-19
 - setRowPrefetch() method, 5-19

- oracle.jdbc.OracleTypes class, 5-22, 12-24
- oracle.jdbc.pool package, 14-16, 16-5
- oracle.jdbc.StructMetaData, 8-62
- oracle.jdbc.StructMetaData interface, 8-61
- oracle.jdbc.xa package and subpackages, 15-5
- OracleJTADatasource class, 17-25
- OracleOCIConnection class, 16-4
- OracleOCIConnectionPool class, 16-2, 16-4
- OracleOCIFailover interface, 16-6
- OraclePooledConnection class, 14-15, 14-16, 16-2
- OraclePooledConnection method
 - definitions, 14-16
- OraclePooledConnection object, 13-2
- OraclePreparedStatement interface, 5-19
 - getOraclePlsqlIndexTable() method, 16-22
 - getPlsqlIndexTable() method, 16-22
 - registerIndexTableOutParameter() method, 16-22
 - setPlsqlIndexTable() method, 16-22
 - setTIMESTAMP(), 5-14
 - setTIMESTAMPPLTZ(), 5-14
 - setTIMESTAMPTZ(), 5-14
- OraclePreparedStatement object, 13-3, 13-4
- OracleResultSet interface, 5-21
 - getXXX() methods, 6-7
- OracleResultSetCache interface, 11-6
- OracleResultSetMetaData interface, 5-22
- OracleServerDriver class
 - defaultConnection() method, 18-28
- oracle.sql datatype classes, 5-7
- oracle.sql package
 - data conversions, 6-2
 - described, 5-7
- oracle.sql.ARRAY class, 10-2
 - and nested tables, 5-12
 - and VARRAYs, 5-12
 - createDescriptor() method, 10-14
 - getArray() method, 10-6
 - getArrayType() method, 10-14
 - getAutoBuffering() method, 10-9
 - getBaseType() method, 10-6
 - getBaseTypeName() method, 10-6
 - getDescriptor() method, 10-6
 - getJavaSQLConnection() method, 10-6, 10-15
 - getMaxLength() method, 10-14
 - getOracleArray() method, 10-6
 - getResultSet() method, 10-6
 - getSQLTypeName() method, 10-6
 - length() method, 10-6
 - methods for Java primitive types, 10-8
 - setAutoBuffering() method, 10-9
 - setAutoIndexing() method, 10-10
- oracle.sql.ArrayDescriptor class
 - getBaseName() method, 10-14
 - getBaseType() method, 10-14
- oracle.sql.BFILE class, 5-12
 - closeFile() method, 7-26
 - getBinaryStream() method, 7-26
 - getBytes() method, 7-26
 - getDirAlias() method, 7-27
 - getName() method, 7-27
 - isFileOpen() method, 7-27
 - length() method, 7-27
 - openFile() method, 7-26
 - position() method, 7-27
- oracle.sql.BLOB class, 5-12
 - getBinaryOutputStream() method, 7-14
 - getBinaryStream() method, 7-14
 - getBufferSize() method, 7-14
 - getBytes() method, 7-14
 - getChunkSize() method, 7-14
 - length() method, 7-14
 - position() method, 7-14
 - putBytes() method, 7-14
- oracle.sql.CHAR class, 18-34
 - getString() method, 5-32
 - getStringWithReplacement() method, 5-32
 - toString() method, 5-32
- oracle.sql.CharacterSet class, 5-30
- oracle.sql.CLOB class, 5-12
 - getAsciiOutputStream() method, 7-15
 - getAsciiStream() method, 7-15
 - getBufferSize() method, 7-15
 - getCharacterOutputStream() method, 7-15
 - getCharacterStream() method, 7-15
 - getChars() method, 7-15
 - getChunkSize() method, 7-16
 - getSubString() method, 7-16
 - length() method, 7-16
 - position() method, 7-16

- putChars() method, 7-16
- putString() method, 7-16
- supported character sets, 7-13
- oracle.sql.datatypes
 - support, 5-9
- oracle.sql.DATE class, 5-13
- oracle.sql.Datum array, 16-26
- oracle.sql.Datum class, described, 5-7
- oracle.sql.NUMBER class, 5-13
- oracle.sql.ORADData, 8-53
- oracle.sql.ORADData interface, 8-21
- oracle.sql.ORADDataFactory, 8-53
- oracle.sql.ORADDataFactory interface, 8-21
- OracleSql.parse() method, 21-14
- oracle.sql.RAW class, 5-13
- oracle.sql.REF class, 5-11, 9-2
 - getBaseTypeName() method, 9-4
 - getValue() method, 9-5
 - setValue() method, 9-5
- oracle.sql.ROWID class, 5-10, 5-15, 5-33
- oracle.sql.STRUCT class, 5-10, 8-4
 - getAutoBuffering() method, 8-9
 - getDescriptor() method, 8-4
 - getJavaSQLConnection() method, 8-4
 - getOracleAttributes() method, 8-4
 - setAutoBuffering() method, 8-9
 - toJDBC() method, 8-4
- oracle.sql.StructDescriptor class, 8-61
 - createDescriptor() method, 8-5
- OracleStatement interface, 5-19
- OracleTypes class, 5-22
- OracleTypes class for typecodes, 5-22
- OracleTypes.CURSOR variable, 5-36
- OracleXAConnection class, 15-7
- OracleXADataSource class, 15-6
- OracleXAResource class, 15-8, 15-9
- OracleXid class, 15-13
- ORADData interface, 5-4
 - additional uses, 8-26
 - advantages, 8-11
 - example program, 20-47
 - Oracle object types, 8-1
 - reading data, 8-23
 - writing data, 8-25
- othersDeletesAreVisible() method (database meta

- data), 11-28
- othersInsertsAreVisible() method (database meta data), 11-28
- othersUpdatesAreVisible() method (database meta data), 11-28
- OUT parameter mode, 16-24, 16-25
 - code example, 20-11
- outer joins, SQL92 syntax, 21-13
- ownDeletesAreVisible() method (database meta data), 11-27
- ownInsertsAreVisible() method (database meta data), 11-28
- ownUpdatesAreVisible() method (database meta data), 11-27

P

- parameter modes
 - IN, 16-22
 - code example, 20-11
 - IN OUT, 16-24
 - code example, 20-11
 - OUT, 16-24, 16-25
 - code example, 20-11
- password connection property, 3-6
- password, specifying, 3-5
- PATH variable, specifying, 2-7
- PDA, A-13
- performance enhancements, standard vs. Oracle, 4-5
- performance extensions
 - defining column types, 12-23
 - prefetching rows, 12-20
 - TABLE_REMARKS reporting, 12-26
- performance optimization, 19-6
- Personal Digital Assistant (PDA), A-13
- PL/SQL
 - IN parameter, 8-59
 - OUT parameters, 8-60
 - restrictions, 19-9
 - space padding, 19-8
 - stored procedures, 3-31
- PL/SQL index-by tables
 - mapping, 16-25
 - scalar datatypes, 16-21

PL/SQL types

- corresponding JDBC types, 16-21
- limitations, 21-16
- PoolConfig() method, 16-7
- pooled connections
 - Oracle implementation, 14-15
 - standard interface, 14-15
- populate() method, A-10
- position() method, 7-14, 7-16, 7-27
- positioning in result sets, 11-2
- prefetching rows, 12-20
 - suggested default, 12-23
- prepare a distributed transaction branch, 15-11
- prepareCall(), 13-5
- prepareCall() method, 5-18, 13-8, 13-9, 13-10
- prepareCallWithKey(), 13-5
- prepareCallWithKey() method, 13-10, 13-11
- prepared statement
 - passing BFILE locator, 7-21
 - passing LOB locators, 7-5
 - using setObject() method, 6-12
 - using setObject() method, 6-12
- PreparedStatement object
 - creating, 3-12
- prepareStatement(), 13-5
- prepareStatement() method, 5-18, 13-8, 13-9, 13-10
 - code example, 13-8
- prepareStatementWithKey(), 13-5
- prepareStatementWithKey() method, 13-10, 13-11
- previous() method (result set), 11-15
- printStackTrace() method (SQLException), 3-34
- PrintWriter for a data source, 14-11
- put() method
 - for Properties object, 3-9
 - for type maps, 8-13
- putBytes() method, 7-14
- putChars() method, 7-16
- putString() method, 7-16

Q

- query, executing, 3-11

R

- RAW class, 5-13
- RDBMS, 1-7
- read-only result set concurrency type, 11-4
- readSQL() method, 8-15, 8-16, 8-54, 8-61
 - implementing, 8-16
- REF class, 5-11
- REF CURSORS, 5-35
 - example program, 20-38
 - materialized as result set objects, 5-35
- refetching rows into a result set, 11-26, 11-29
- refreshRow() method (result set), 11-26
- registerDriver() method, 5-17
- registerIndexTableOutParameter() method, 16-22, 16-24
 - arguments
 - int elemMaxLen, 16-24
 - int elemSqlType, 16-24
 - int maxLen, 16-24
 - int paramIndex, 16-24
 - code example, 16-25
- registering Oracle JDBC drivers, class for, 5-17
- registerOutParameter() method, 5-21, 6-13, 8-61
- Relational Database Management System (RDBMS), 1-7
- relative positioning in result sets, 11-2
- relative() method (result set), 11-14
- remarksReporting connection property, 3-7
- remarksReporting flag, 12-20
- Remote Method Invocation (RMI), A-12
- removeConnectionEventListener method (connection cache), 14-23
- resource managers, 15-3
- result set
 - auto-commit mode, 19-6
 - getting BFILE locators, 7-20
 - getting LOB locators, 7-4
 - metadata, 5-22
 - Oracle extensions, 6-3
 - using getObject() method, 6-5
- result set enhancements
 - positioning result sets, 11-13
- result set enhancements
 - concurrency types, 11-4

- downgrade rules, 11-11
- fetch size, 11-24
- limitations, 11-10
- Oracle scrollability requirements, 11-5
- Oracle updatability requirements, 11-5
- positioning, 11-2
- processing result sets, 11-16
- refetching rows, 11-26, 11-29
- result set types, 11-3
- scrollability, 11-2
 - seeing external changes, 11-28
 - seeing internal changes, 11-27
 - sensitivity to database changes, 11-2
 - specifying scrollability, updatability, 11-8
 - summary of methods, 11-32
 - summary of visibility of changes, 11-30
 - updatability, 11-4
 - updating result sets, 11-18
 - visibility vs. detection of external changes, 11-29
- result set fetch size, 11-24
- result set methods, JDBC 2.0, 11-32
- result set object
 - closing, 3-12
- result set types for scrollability and sensitivity, 11-3
- result set, processing, 3-11
- ResultSet class, 3-11
- ResultSet() method, 10-10
- ResultSetMetaData class, 8-62
- return types
 - for getXXX() methods, 6-7
 - getObject() method, 6-6
 - getOracleObject() method, 6-6
- return values
 - casting, 6-10
- reusePooledConnection() method, 14-24
- RMI, A-12
- roll back a distributed transaction branch, 15-12
- roll back changes to database, 3-13
- rollback method, 17-3, 17-4, 17-16, 17-19
- RollbackException, 17-4
- row prefetching, 12-20
 - and data streams, 3-30
- ROWID class, 5-15

- CursorName methods, 21-16
 - defined, 5-33
- ROWID, use for result set updates, 11-5

S

- scalar functions, SQL92 syntax, 21-12
- schema naming conventions, 5-5
- scrollability in result sets, 11-2
- scrollable result sets
 - creating, 11-8
 - fetch direction, 11-17
 - implementation of scroll-sensitivity, 11-30
 - positioning, 11-13
 - processing backward/forward, 11-16
 - refetching rows, 11-26, 11-29
 - scroll-insensitive result sets, 11-3
 - scroll-sensitive result sets, 11-3
 - seeing external changes, 11-28
 - visibility vs. detection of external changes, 11-29
- scroll-sensitive result sets
 - limitations, 11-10
- security
 - authentication, 18-10
 - encryption, 18-11
 - integrity, 18-11
 - Oracle Advanced Security support, 18-9
 - overview, 18-9
- SecurityException, 17-4
- SELECT statement
 - to retrieve object references, 9-6
 - to select LOB locator, 7-12
- sendBatch() method, 12-7, 12-9
- sensitivity in result sets to database changes, 11-2
- serialization
 - ArrayDescriptor object, 10-15
 - definition of, 8-6, 10-15
 - StructDescriptor object, 8-6
- server-side internal driver
 - connection to database, 18-27
- server-side Thin driver, described, 1-9
- sess_sh tool, 14-9
- session context, 1-15
 - for KPRB driver, 18-31

- setAsciiStream() method, 6-16
- setAutoBuffering() method
 - of the oracle.sql.ARRAY class, 10-9
 - of the oracle.sql.STRUCT class, 8-9
- setAutoCommit() method, 19-6
- setAutoIndexing() method, 10-10
 - direction parameter values
 - ARRAY.ACCESS_FORWARD, 10-10
 - ARRAY.ACCESS_REVERSE, 10-10
 - ARRAY.ACCESS_UNKNOWN, 10-10
- setBFILE() method, 7-21
- setBinaryStream() method, 6-16
- setBLOB() method, 7-5
- setBlob() method, JDK 1.1.x, 7-5
- setBlob() method, JDK 1.2.x, 7-5
- setBytes() limitations, using streams to avoid, 3-30
- setCacheScheme() method (connection cache), 14-28
- setCharacterStream() method, 6-16
- setCLOB() method, 7-5
- setClob() method, 1.1.x, 7-5
- setClob() method, JDK 1.2.x, 7-5
- setConnection() method
 - ArrayDescriptor object, 10-15
 - StructDescriptor object, 8-6
- setConnectionPoolDataSource method (connection cache), 14-26
- setCursorName() method, 21-16
- setDataSource() method (connection event listener), 14-29
- setDate() method, 6-16
- setDefaultExecuteBatch() method, 5-18, 12-5
- setDefaultRowPrefetch() method, 5-19, 12-21
- setDisableStmtCaching() method, 13-8
- setEscapeProcessing() method, 21-10
- setExecuteBatch() method, 5-20, 12-6
- setFetchSize() method, 11-24
- setFixedCHAR() method, 6-17
- setFormOfUse() method, 5-28
- setMaxFieldSize() method, 12-25, 19-8
- setNull() method, 5-20, 5-21, 6-13
- setObeject() method, 6-11
- setObject() method
 - for BFILES, 7-21
 - for BLOBs and CLOBs, 7-5
 - for object references, 9-8
 - for ORADData objects, 8-23
 - for STRUCT objects, 8-8
 - to write object data, 8-26
 - using in prepared statements, 6-12
- setOracleObject() method, 5-20, 5-21, 6-11
 - for BFILES, 7-21
 - for BLOBs and CLOBs, 7-5
 - using in prepared statements, 6-12
- setORADData() method, 5-20, 8-22, 8-26
- setPlsqlIndexTable() method, 16-22
 - arguments
 - int curLen, 16-23
 - int elemMaxLen, 16-23
 - int elemSqlType, 16-23
 - int maxLen, 16-22
 - int paramIndex, 16-22, 16-25
 - Object arrayData, 16-22
 - code example, 16-23
- setPoolConfig() method, 16-7
- setREF() method, 9-8
- setRemarksReporting() method, 5-19, 12-27
- setResultSetCache() method, 11-6
- setRollbackOnly method, 17-5
- setRowPrefetch() method, 5-19, 12-21
- setStmtCacheSize() method, 13-6, 16-11
 - code example, 13-7
- setString() limitations, using streams to avoid, 3-30
- setString() method
 - to bind ROWIDs, 5-33
- setTime() method, 6-16
- setTimestamp() method, 6-17
- setTransactionIsolation() method, 5-18, 19-14
- setTransactionTimeout method, 17-5
- setTypeMap() method, 5-18
- setUnicodeStream() method, 6-16
- setValue() method, 9-5
- setXXX() methods
 - Oracle extended properties, 14-6
 - setXXX() methods, for empty LOBs, 7-17
 - setXXX() methods, for specific datatypes, 6-12
 - signed applets, 1-12
- Solaris
 - shared libraries
 - libheteroxa9_g.so, 16-19

- libheteroxa9.so, 16-19
- SQL
 - data converting to Java datatypes, 6-2
 - primitive types, 5-7
 - structured types, 5-7
 - types, constants for, 5-22
- SQL engine
 - relation to the KPRB driver, 18-27
- SQL syntax (Oracle), 21-10
- SQL*Plus, 8-55, 8-56, 8-59
- SQL92 syntax, 21-10
 - function call syntax, 21-14
 - LIKE escape characters, 21-13
 - outer joins, 21-13
 - scalar functions, 21-12
 - time and date literals, 21-10
 - translating to SQL example, 21-14
- SQLData interface, 5-4
 - advantages, 8-11
 - described, 8-15
 - example program, 20-43
 - Oracle implementation, 5-26
 - Oracle object types, 8-1
 - reading data from Oracle objects, 8-17
 - using with type map, 8-15
 - writing data from Oracle objects, 8-20
- SQLInput interface, 8-15
 - described, 8-16
- SQLInput streams, 8-16
- SQLJ
 - advantages over JDBC, 1-3
 - guidelines for using, 1-4
- SQLJ object type, 8-52
- SQLNET.ORA
 - parameters for tracing, 19-11
- SQLOutput interface, 8-15
 - described, 8-16
- SQLOutput streams, 8-17
- SQLWarning class, limitations, 21-17
- start a distributed transaction branch, 15-10
- statement caching
 - explicit, 13-2
 - code example, 20-82
 - definition of, 13-4
 - null data, 13-11
 - implicit, 13-2
 - code example, 20-79
 - concept graphic, 13-3
 - definition of, 13-2
 - Least Recently Used (LRU) scheme, 13-4
 - statement methods, JDBC 2.0 result sets, 11-35
- Statement object
 - closing, 3-12
 - creating, 3-11
- statements
 - Oracle extensions, 6-3
- static SQL, 1-2
- stored procedures
 - Java, 3-32
 - PL/SQL, 3-31
- stream data, 3-19, 7-6
 - CHAR columns, 3-24
 - closing, 3-28
 - example, 3-21
 - external files, 3-27
 - LOBs, 3-27
 - LONG columns, 3-19
 - LONG RAW columns, 3-19
 - multiple columns, 3-25
 - precautions, 3-28
 - RAW columns, 3-24
 - row prefetching, 3-30
 - UPDATE/COMMIT statements, 7-8
 - use to avoid setBytes() and setString()
 - limitations, 3-30
 - VARCHAR columns, 3-24
- stream data column
 - bypassing, 3-26
- STRUCT class, 5-10
- STRUCT descriptor, 8-4, 8-5
- STRUCT object, 5-10
 - attributes, 5-10
 - creating, 8-4, 8-5
 - embedded object, 8-7
 - nested objects, 5-11
 - retrieving, 8-7
 - retrieving attributes as oracle.sql types, 8-8
- StructDescriptor object
 - creating, 8-5
 - deserialization, 8-6

- get methods, 8-5
- serialization, 8-6
- setConnection() method, 8-6

StructMetaData interface, 8-62

SystemException, 17-4

T

TABLE_REMARKS columns, 12-20

TABLE_REMARKS reporting

- restrictions on, 12-26

TAF, definition of, 16-16

TCP/IP protocol, 1-7, 3-10

Thin driver

- applets, 1-12, 18-16
- applications, 1-12
- CHAR/VARCHAR2 NLS size restrictions, 18-6
- described, 1-7
- NLS considerations, 18-4
- server-side, described, 1-9

time and date literals, SQL92 syntax, 21-10

tnsEntry, 14-5, 16-19

TNSNAMES entries, 3-4

toDatum() method, 8-54

- applied to ORADData objects, 8-11
- applied to ORADate objects, 8-21
- called by setORADData() method, 8-26

toJDBC() method, 8-4

toJdbc() method, 5-10

toString() method, 5-32

trace facility, 19-11

trace parameters

- client-side, 19-11
- server-side, 19-12

tracing with a data source, 14-11

transaction

- client-side demarcation, 17-3, 17-15
- demarcation, 17-3
- enlisting resources, 17-5
- limitations, 17-8
- two-phase commit, 17-7, 17-21

transaction branch, 15-2

transaction branch ID component, 15-13

Transaction class, 17-3

transaction context, 1-15

- for KPRB driver, 18-31

transaction IDs (distributed transactions), 15-5

transaction managers, 15-2

TransactionManager class, 17-3

transactions

- global, 17-3
- overview, 17-2

Transparent Application Failover (TAF), definition of, 16-16

TTC error messages, listed, B-14

TTC protocol, 1-7, 1-8

two-phase commit, 17-21

type map, 5-4, 6-4

- adding entries, 8-13
- and STRUCTS, 8-15
- creating a new map, 8-14
- used with arrays, 10-18
- used with SQLData interface, 8-15
- using with arrays, 10-25

type map (SQL to Java), 8-10

type mapping

- BigDecimal mapping, 8-47
- JDBC mapping, 8-46
- object JDBC mapping, 8-46
- Oracle mapping, 8-46

type mappings

- JPublisher options, 8-45

type maps

- relationship to database connection, 18-29

TYPE_FORWARD_ONLY result sets, 11-8

TYPE_SCROLL_INSENSITIVE result sets, 11-8

TYPE_SCROLL_SENSITIVE result sets, 11-8

typecodes, Oracle extensions, 5-22

U

unicode data, 5-28

UNIX, 14-9

updatability in result sets, 11-4

updatable result set concurrency type, 11-4

updatable result sets

- creating, 11-8
- DELETE operations, 11-18
- INSERT operations, 11-21
- limitations, 11-10

- refetching rows, 11-26, 11-29
- seeing internal changes, 11-27
- update conflicts, 11-23
- UPDATE operations, 11-19
- update batching
 - overview, Oracle vs. standard model, 12-2
 - overview, statements supported, 12-3
- update batching (Oracle model)
 - batch value, checking, 12-7
 - batch value, overriding, 12-7
 - committing changes, 12-8
 - connection batch value, setting, 12-5
 - connection vs. statement batch value, 12-4
 - default batch value, 12-5
 - disable auto-commit, 12-4
 - example, 12-9
 - limitations and characteristics, 12-5
 - overview, 12-4
 - statement batch value, setting, 12-6
 - stream types not allowed, 12-5
 - update counts, 12-9
- update batching (standard model)
 - adding to batch, 12-11
 - clearing the batch, 12-14
 - committing changes, 12-14
 - error handling, 12-16
 - example, 12-15
 - executing the batch, 12-13
 - intermixing batched and non-batched, 12-17
 - overview, 12-10
 - sample application, 20-69
 - stream types not allowed, 12-11
 - update counts, 12-15
 - update counts upon error, 12-17
- update conflicts in result sets, 11-23
- update counts
 - Oracle update batching, 12-9
 - standard update batching, 12-15
 - upon error (standard batching), 12-17
- UPDATE in a result set, 11-19
- updateRow() method (result set), 11-20
- updatesAreDetected() method (database meta data), 11-29
- updateXXX() methods (result set), 11-19, 11-21
- updateXXX() methods for empty LOBs, 7-17

- updating result sets, 11-18
- url, 14-6
- user connection property, 3-6
- userid, specifying, 3-5
- UserTransaction
 - bind in namespace, 17-12
- UserTransaction interface, 17-3
 - begin method, 17-3
 - commit method, 17-4
 - getStatus method, 17-5
 - rollback method, 17-4
 - setRollbackOnly method, 17-5
 - setTransactionTimeout method, 17-5
- UserTransaction object
 - begin method, 17-3, 17-16, 17-19
 - binding in namespace, 17-18
 - commit method, 17-3, 17-16, 17-19
 - rollback method, 17-3, 17-16, 17-19
- using, 18-18

V

- VARCHAR2 columns, 19-8
 - NLS size restrictions, Thin, 18-6
- varrays
 - example program, 20-35

W

- WIDTH, parameter for APPLET tag, 18-25
- window, scroll-sensitive result sets, 11-30
- writeSQL() method, 8-15, 8-17, 8-54, 8-61
 - implementing, 8-16

X

XA

- connection implementation, 15-7
- connections (definition), 15-4
- data source implementation, 15-6
- data sources (definition), 15-3
- definition, 15-2
- error handling, 15-16
- example of implementation, 15-18
- exception classes, 15-15

- Oracle optimizations, 15-17
- Oracle transaction ID implementation, 15-13
 - resource implementation, 15-8
 - resources (definition), 15-4
 - sample application (suspend/resume), 20-105
 - sample application (two-phase commit), 20-1, 20-109
 - transaction ID interface, 15-13
- XAException, 15-13
- Xids, 15-13