## Contents

Send Us Your Comments ................................................................................................................ix

Preface .................................................................................................................................................. xi

- Intended Audience ......................................................................................................................... xi
- Manual Structure ............................................................................................................................ xi
- Related Documentation .................................................................................................................. xii
- Conventions Used in this Manual ................................................................................................... xv

### 1 Overview

What is JDBC? .................................................................................................................................. 1-2

- JDBC versus SQLJ .......................................................................................................................... 1-2
  - Advantages of SQLJ over JDBC for Static SQL ......................................................................... 1-3
  - General Guidelines for using JDBC and SQLJ ............................................................................ 1-3

Basic Driver Architecture .................................................................................................................. 1-4

- JDBC Thin Client-Side Driver Architecture .................................................................................. 1-5
- JDBC OCI Client-Side Driver Architecture .................................................................................... 1-5
- JDBC Server Driver Architecture ................................................................................................... 1-6

Oracle Extensions to the JDBC Standard .......................................................................................... 1-6

Supported JDK and JDBC Versions .................................................................................................. 1-6

JDBC and the Oracle Application Server ......................................................................................... 1-6

JDBC and IDEs .................................................................................................................................. 1-7

### 2 Getting Started

Oracle JDBC Drivers ......................................................................................................................... 2-2
Introducing the Oracle JDBC Drivers ................................................................. 2-2
Choosing the Appropriate Driver ..................................................................... 2-4
Requirements and Compatibilities for Oracle JDBC Drivers ....................... 2-5
Verifying a JDBC Client Installation ................................................................. 2-6
Check Installed Directories and Files............................................................... 2-6
Check the Environment Variables .................................................................... 2-7
Make Sure You Can Compile and Run Java ..................................................... 2-7
Determining 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
Importing Packages ........................................................................................... 3-2
Registering the JDBC Drivers .......................................................................... 3-3
Opening a Connection to a Database ............................................................... 3-3
Creating a Statement Object ........................................................................... 3-8
Executing a Query and Returning a Result Set Object ..................................... 3-8
Processing the Result Set ................................................................................ 3-9
Closing the Result Set and Statement Objects ................................................ 3-9
Closing the Connection ..................................................................................... 3-10

Sample: Connecting, Querying, and Processing the Results ......................... 3-10

Datatype Mappings .......................................................................................... 3-11
Oracle JDBC Extension Types .......................................................................... 3-12
Using Java Streams in JDBC .......................................................................... 3-14
Streaming LONG or LONG RAW Columns .................................................... 3-14
Streaming CHAR, VARCHAR, or RAW Columns ............................................. 3-19
Data Streaming and Multiple Columns ............................................................ 3-20
Streaming and Row Prefetching ...................................................................... 3-23
Closing a Stream ............................................................................................. 3-23
Streaming LOBs and External Files ................................................................. 3-23

Using Stored Procedures in JDBC Programs .................................................. 3-24
PL/SQL Stored Procedures .............................................................................. 3-24
Java Stored Procedures .................................................................................... 3-25
Error Messages and JDBC .............................................................................. 3-25
Server-Side Basics .......................................................................................... 3-26
Session and Transaction Context ................................................................. 3-26
Connecting to the Database ........................................................................... 3-26

Application Basics versus Applet Basics ...................................................... 3-27
  Application Basics ......................................................................................... 3-27
  Applet Basics ............................................................................................... 3-27

4 Oracle Extensions

Introduction to Oracle Extensions ........................................................................ 4-2
Oracle JDBC Packages and Classes ................................................................. 4-6
  Classes of the oracle.jdbc2 Package .............................................................. 4-6
  Classes of the oracle.sql Package ................................................................. 4-7
  Classes of the oracle.jdbc.driver Package ..................................................... 4-22
Data Access and Manipulation: Oracle Types vs. Java Types ......................... 4-32
  Data Conversion Considerations .................................................................... 4-32
  Using Result Set and Statement Extensions .................................................. 4-33
  Comparing get and set Methods for oracle.sql.* Format with Java Format .... 4-34
  Using Result Set Meta Data Extensions .......................................................... 4-44

Working with LOBs ......................................................................................... 4-45
  Getting BLOB and CLOB Locators ................................................................. 4-46
  Passing BLOB and CLOB Locators ................................................................. 4-47
  Reading and Writing BLOB and CLOB Data ................................................... 4-48
  Creating and Populating a BLOB or CLOB Column ....................................... 4-52
  Accessing and Manipulating BLOB and CLOB Data ...................................... 4-54
  Getting BFILE Locators .................................................................................. 4-55
  Passing BFILE Locators ................................................................................ 4-56
  Reading BFILE Data ....................................................................................... 4-57
  Creating and Populating a BFILE Column .................................................... 4-58
  Accessing and Manipulating BFILE Data ...................................................... 4-60

Working with Oracle Object Types ................................................................. 4-62
  Using Default Java Classes for Oracle Objects ............................................. 4-62
  Creating Custom Java Classes for Oracle Objects ......................................... 4-65
  Using JPublisher with JDBC ........................................................................... 4-82

Working with Oracle Object References ....................................................... 4-83
  Retrieving an Object Reference ..................................................................... 4-84
  Passing an Object Reference to a Callable Statement .................................... 4-85
5  Advanced Topics

Using NLS .................................................................................................................. 5-2
How JDBC Drivers Perform NLS Conversions ............................................................. 5-2
NLS Restrictions ........................................................................................................... 5-5

Working with Applets ............................................................................................... 5-7
Coding Applets ........................................................................................................... 5-7
Connecting an Applet to a Database .......................................................................... 5-9
Using Applets with Firewalls ...................................................................................... 5-14
Packaging Applets ...................................................................................................... 5-17
Specifying an Applet in an HTML Page ....................................................................... 5-19
Browser Security and JDK Version Considerations ..................................................... 5-20

JDBC on the Server: the Server Driver ...................................................................... 5-22
Connecting to the Database with the Server Driver ..................................................... 5-22
Session and Transaction Context for the Server Driver ............................................... 5-23
Testing JDBC on the Server ........................................................................................ 5-24
Server Driver Support for NLS ..................................................................................... 5-25

Embedded SQL92 Syntax ........................................................................................... 5-26
Time and Date Literals ............................................................................................... 5-26
Scalar Functions ........................................................................................................... 5-28
LIKE Escape Characters ............................................................................................. 5-29
Outer Joins .................................................................................................................... 5-29
Function Call Syntax .................................................................................................. 5-30
SQL92 to SQL Syntax Example .................................................................................... 5-30
6  Coding Tips and Troubleshooting

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC and Multi-Threading</td>
<td>6-2</td>
</tr>
<tr>
<td>Performance Optimization</td>
<td>6-5</td>
</tr>
<tr>
<td>Disabling Auto-Commit Mode</td>
<td>6-5</td>
</tr>
<tr>
<td>Prefetching Rows</td>
<td>6-6</td>
</tr>
<tr>
<td>Batching Updates</td>
<td>6-6</td>
</tr>
<tr>
<td>Common Problems</td>
<td>6-6</td>
</tr>
<tr>
<td>Space Padding for CHAR Columns Defined as OUT or IN/OUT Variables</td>
<td>6-7</td>
</tr>
<tr>
<td>Memory Leaks and Running Out of Cursors</td>
<td>6-7</td>
</tr>
<tr>
<td>Boolean Parameters in PL/SQL Stored Procedures</td>
<td>6-7</td>
</tr>
<tr>
<td>Opening More Than 16 OCI Connections for a Process</td>
<td>6-8</td>
</tr>
<tr>
<td>Basic Debugging Procedures</td>
<td>6-9</td>
</tr>
<tr>
<td>Trapping Exceptions</td>
<td>6-9</td>
</tr>
<tr>
<td>Logging JDBC Calls</td>
<td>6-10</td>
</tr>
<tr>
<td>Net8 Tracing to Trap Network Events</td>
<td>6-10</td>
</tr>
<tr>
<td>Using Third Party Tools</td>
<td>6-13</td>
</tr>
<tr>
<td>Transaction Isolation Levels and the Oracle Server</td>
<td>6-13</td>
</tr>
</tbody>
</table>

7  Sample Applications

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Applications for Basic JDBC Features</td>
<td>7-2</td>
</tr>
<tr>
<td>Streaming Data</td>
<td>7-2</td>
</tr>
<tr>
<td>Sample Applications for JDBC 2.0-Compliant Oracle Extensions</td>
<td>7-4</td>
</tr>
<tr>
<td>LOB Sample</td>
<td>7-4</td>
</tr>
<tr>
<td>BFILE Sample</td>
<td>7-10</td>
</tr>
<tr>
<td>Sample Applications for Other Oracle Extensions</td>
<td>7-14</td>
</tr>
<tr>
<td>REF CURSOR Sample</td>
<td>7-14</td>
</tr>
<tr>
<td>Array Sample</td>
<td>7-16</td>
</tr>
<tr>
<td>Creating Customized Java Classes for Oracle Objects</td>
<td>7-20</td>
</tr>
<tr>
<td>SQLData Sample</td>
<td>7-20</td>
</tr>
<tr>
<td>CustomDatum Sample</td>
<td>7-26</td>
</tr>
<tr>
<td>Creating Signed Applets</td>
<td>7-31</td>
</tr>
<tr>
<td>JDBC versus SQLJ Sample Code</td>
<td>7-38</td>
</tr>
<tr>
<td>SQL Program to Create Tables and Objects</td>
<td>7-39</td>
</tr>
<tr>
<td>JDBC Version of the Sample Code</td>
<td>7-41</td>
</tr>
<tr>
<td>SQLJ Version of the Sample Code</td>
<td>7-44</td>
</tr>
</tbody>
</table>
8 Reference Information

Valid SQL-JDBC Datatype Mappings .............................................................. 8-2
Supported SQL and PL/SQL Datatypes ......................................................... 8-4
NLS Character Set Support ........................................................................... 8-8
Related Information ...................................................................................... 8-8
  Oracle JDBC Drivers and SQLJ ................................................................. 8-8
  Java Technology ...................................................................................... 8-8
  Signed Applets ...................................................................................... 8-9

A JDBC Error Messages
Send Us Your Comments

JDBC Developer’s Guide and Reference, Release 8.1.5
Part No. A64685-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

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 — jpgcomnt@us.oracle.com
- FAX - 650-506-7225.  Attn: Java Products Group, Information Development Manager
- Postal service:
  Oracle Corporation
  Information Development Manager
  500 Oracle Parkway, Mailstop 4op978
  Redwood Shores, CA  94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

If you have problems with the software, please contact your local Oracle World Wide Support Center.
This preface contains these sections:

- Intended Audience
- Manual Structure
- Related Documentation
- Conventions Used in this Manual

**Intended Audience**

This manual assumes that you are an experienced programmer and that you understand Oracle databases, the SQL and Java programming languages, and the principles of JDBC.

**Manual Structure**

The *JDBC Developers Guide and Reference* contains eight chapters and one appendix:

- **Chapter 1, "Overview"**
  This chapter provides an overview of the Oracle implementation of JDBC and the Oracle JDBC driver architecture.

- **Chapter 2, "Getting Started"**
  This chapter 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.
This manual contains references to the following Oracle publications:

- **Oracle8i 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 `REF` types, then you are required to have Java classes that correspond to these types. JPublisher helps you do this by creating the mapping between object types and Java classes, and between object attribute types and their corresponding Java types.
Oracle8i SQLJ Developer’s Guide and Reference

This book describes the use of SQLJ to embed static SQL operations directly into Java code. Both standard SQLJ features and Oracle-specific SQLJ features are described.

Oracle8i Java Stored Procedures Developer’s Guide

This book describes Java stored procedures, which lets Java programmers access the Oracle RDBMS. 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.

Oracle8i Enterprise JavaBeans and CORBA Developer’s Guide

This manual describes the Oracle extensions to the JavaBeans and CORBA specifications.

Net8 Administrator’s Guide

Refer to this manual for more information about ANO (Advanced Network Option), the Oracle8 Connection Manager, and about Net8 network administration in general.

Oracle8i Error Messages

Refer to this document set for more information on error messages that can be passed by the Oracle Database and the Oracle JDBC drivers.

Oracle8i National Language Support Guide

Refer to this manual for more information on NLS environment variables, character sets, territories, and locale data. In addition, it contains an overview of common NLS issues, some typical scenarios, and some NLS considerations for OCI and SQL programmers.

Oracle8i Application Developer’s Guide - Large Objects (LOBs) and the Oracle8i Application Developer’s Reference - Packages

These books describe how to access and manipulate large objects (LOBs) using PL/SQL code and the DBMS_LOB package.

Oracle8i SQL Reference

This reference contains a complete description of the content and syntax of the Structured Query Language (SQL) used to manage information in an Oracle database.
PL/SQL User’s Guide and Reference

PL/SQL is Oracle’s procedural extension to SQL. An advanced fourth-generation programming language (4GL), PL/SQL offers seamless SQL access, tight integration with the Oracle server and tools, portability, security, and modern software engineering features such as data encapsulation, overloading, exception handling, and information hiding. This guide explains all the concepts behind PL/SQL and illustrates every facet of the language.

Oracle8i Application Server documentation

Refer to this documentation for more information on how the Oracle8i Application Server supports JDBC.

Oracle8 JDeveloper Suite documentation

Refer to this documentation for more information on how Oracle8 JDeveloper Suite supports JDBC.
Conventions Used in this Manual

Solaris syntax is used in this book, 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:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . .</td>
<td>Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.</td>
</tr>
<tr>
<td>. . .</td>
<td>Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted.</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Angle brackets enclose user-supplied names.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets enclose optional clauses from which you can choose one or none.</td>
</tr>
</tbody>
</table>
This chapter provides an overview of the Oracle implementation of JDBC and contains these topics:

- What is JDBC?
- JDBC versus SQLJ
- Basic Driver Architecture
- Oracle Extensions to the JDBC Standard
- Supported JDK and JDBC Versions
- JDBC and the Oracle Application Server
- JDBC and IDEs
What is JDBC?

JDBC (Java Database Connectivity) is a standard Java interface for connecting to relational databases from Java. 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 the standard JDBC API, Oracle drivers have extensions to properties, types, and 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 of 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 provides 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 that are known at the time the program is written, it can also inter-operate 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 Oracle8i 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. Some advantages you gain in using SQLJ over JDBC for static SQL statements are:

- **SQLJ source programs are smaller than equivalent JDBC programs because SQLJ provides a shorter syntax.**
- **SQLJ uses database connections to type-check static SQL code. JDBC, being a completely dynamic API, does not do any type checking until run-time.**
- **SQLJ programs allow direct embedding of Java bind expressions within SQL statements. JDBC requires separate get and/or set call statements for each bind variable and specifies the binding by position number.**
- **SQLJ provides strong typing of query outputs and return parameters and allows type-checking on calls. JDBC passes values to and from SQL without compile-time type checking.**
- **SQLJ provides simplified rules for calling SQL stored procedures and functions.**

### General Guidelines for using JDBC and SQLJ

Use SQLJ to write your program when:

- 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.

Use JDBC to write your program when:

- your program uses dynamic SQL. For example, you have a program that builds queries on-the-fly or has an interactive 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.

Basic Driver Architecture

This section has the following subsections:

- JDBC Thin Client-Side Driver Architecture
- JDBC OCI Client-Side Driver Architecture
- JDBC Server Driver Architecture

Figure 1–1 illustrates the driver-database architecture for the JDBC Thin, OCI, and Server drivers.
Basic Driver Architecture

Figure 1–1 Driver-Database Architecture

JDBC Thin Client-Side Driver Architecture

The Oracle JDBC Thin driver is a Type IV driver that is targeted to applet developers. This driver is written in 100% Pure Java and complies with the JDBC 1.22 standard.

For communicating with the database, the driver includes an equivalent implementation of Oracle’s TTC presentation protocol and Net8 session protocol in Java. Both of these protocols are lightweight implementation versions of their counterparts on the server. The Net8 protocol runs over TCP/IP only. To use this driver, it is not necessary to install any Oracle-specific software on the client.

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.

JDBC OCI Client-Side Driver Architecture

The JDBC OCI driver is a Type II driver that is targeted to client-server Java applications programmers and Java-based middle-tier developers. The JDBC OCI driver converts JDBC invocations to calls to the Oracle Call Interface (OCI). These calls are then sent over Net8 to the Oracle database server.
The JDBC OCI driver is written in a combination of Java and C because it must make calls to the OCI libraries. The driver requires the presence of the OCI libraries, Net8, CORE libraries, and other necessary files on each client machine or middle-tier application server on which it is installed.

**JDBC Server Driver Architecture**

The JDBC Server driver allows Java programs that use the Oracle 8.1.5 Java Virtual Machine (VM) and run inside the database to communicate with the SQL engine. The Server driver, the Java VM, the database, the KPRB (server-side) C library, and the SQL engine all run within the same address space. There are no network round-trips involved. The programs access the SQL engine by using function calls.

**Oracle Extensions to the JDBC Standard**

The Oracle JDBC drivers support many of the features described in the JDBC 2.0 standard. This support is provided in the form of Oracle-defined extensions for Oracle datatypes, object types, and their mappings to Java. For more information on these extensions, see Chapter 4, "Oracle Extensions".

**Supported JDK and JDBC Versions**

Oracle’s JDBC drivers, release 8.1.5, support the JDK versions 1.0.2 and 1.1.x. They also comply with JDBC version 1.22 and, in addition, implement most of the features of JDBC version 2.0.

---

**Note:** There are special considerations for using the Thin driver with JDK 1.0.2 and 1.1.1 in the context of applets. See "Working with Applets" on page 5-7 for more information on this topic.

---

**JDBC and the Oracle Application Server**

Oracle Application Server is a collection of middleware services and tools that provide a scalable, robust, secure, and extensible platform for distributed, object-oriented applications. Oracle Application Server 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 Oracle Web Application Server versions 3.0 and higher. The Oracle Web Application Server bundles JDBC with its distribution. For more information on the use of JDBC and the Oracle Web Application Server, see your Oracle Web 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, including 100% Pure Java and native Oracle8 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.
This chapter guides you through the basics of testing your installation and configuration and running a simple application. The following topics are discussed:

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

This section has the following subsections:

- Introducing the Oracle JDBC Drivers
- Choosing the Appropriate Driver

Oracle offers two different drivers for client-side use (one of which can also be used in a middle tier) and one for server-side use. Most of the information in the following chapters focuses on the client-side drivers. The server-side driver is described in detail in "JDBC on the Server: the Server Driver" on page 5-22.

Introducing the Oracle JDBC Drivers

This section describes the Oracle JDBC drivers and provides scenarios for how you would use them. Oracle produces JDBC drivers for use in clients and in the server. The client-side drivers can be used in Java applications or Java applets that run either on the client or in the middle tier of a three-tier configuration. The server-side driver provides server-side JDBC support which allows the Java VM to communicate with the SQL engine.

Common Features of Oracle JDBC Drivers

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

- JDBC 1.22
- most of the JDBC 2.0 features
- the same syntax and APIs
- the same Oracle extensions
- full support for multi-threaded applications

The only differences between the drivers are in how they connect to the database and how they transfer data.

JDBC Thin Driver

The Oracle JDBC Thin driver is a 100% Pure Java implementation that complies with the JDBC 1.22 standard. The JDBC Thin driver uses Java Sockets to connect directly to the Oracle Server and is typically used for Java applets in either a two-tier or three-tier configuration, though it can also be used for Java applications. The JDBC Thin driver provides its own implementation of a TCP/IP version of
Oracle’s Net8. Because it is written entirely in Java, this driver is platform-independent. When the JDBC Thin driver is used with an applet, the client browser must have the capability to support Java sockets.

The JDBC Thin driver does not require Oracle software on the client side; it can be downloaded into a browser simultaneously with the Java applet being run. From the client (usually a browser), you select a URL from an HTML page that contains a Java applet tag. The web server downloads the Java applet and the JDBC Thin driver to the client. The JDBC Thin driver then establishes a direct connection to the database server using Java Sockets.

The JDBC Thin driver connects to any Oracle database of version 7.2.3 and higher. The JDBC Thin driver allows a direct connection to the database by emulating Net8 and TTC (the wire protocol used by OCI) on top of Java sockets. The driver supports only TCP/IP protocol and requires a TNS listener to be listening on TCP/IP sockets from the database server.

For a discussion of relevant firewall, browser, and security issues, see "Working with Applets" on page 5-7.

**JDBC OCI Driver**

The JDBC OCI driver provides an implementation of the JDBC interfaces using the Oracle Call Interface (OCI). The OCI driver makes use of the OCI cache, C entry points to OCI, and the OCI library. The use of native methods to call C entry points makes the driver platform-specific. The JDBC OCI driver also requires an Oracle client installation including Net8.

The JDBC OCI driver is compatible with all Oracle versions because it interfaces to Oracle databases through OCI. The driver also supports all installed Net8 adapters, including IPC, named pipes, TCP/IP, and IPX/SPX.

Because the JDBC OCI driver contains C code, it is not suitable for use in applets. However, it is an excellent choice for Java applications or Java middle tiers such as the Oracle Web Application Server. You can use the JDBC OCI driver in these configurations:

- with a Java application running on a client machine in a two-tier configuration
- with a Java application running on a middle tier in a three-tier configuration
- with a Java servlet running on a middle tier in a three-tier configuration
**Oracle JDBC Drivers**

**JDBC Server Driver**

Oracle’s JDBC Server driver is for server-side use only. The Server driver provides server-side JDBC support for any Java program used in the database, Java stored procedure, Enterprise Java Beans (EJB) and for communication with SQL and PL/SQL programs. The Server driver is fully consistent with, and supports the same features and extensions as the client-side drivers. For more information on the server-side driver, see "JDBC on the Server: the Server Driver" on page 5-22.

**Choosing the Appropriate Driver**

Four main considerations that you must bear in mind when choosing which JDBC driver to use for your application or applet are:

- If you are writing an applet, you must use the JDBC Thin driver. JDBC OCI-based driver classes cannot be downloaded to a Web browser, because they call native (C language) methods.

**Notes:**

- JDBC Thin drivers use a subset of the Net8 protocol, written entirely in Java, and connect using the TCP/IP protocol.
- There are other restrictions on applets besides your choice of JDBC driver. For information on these restrictions, see "Browser Security and JDK Version Considerations" on page 5-20.

- If you desire maximum portability, then choose the JDBC Thin driver. You can connect to an Oracle8 data server from either an application or an applet using the JDBC Thin driver.
- If you are writing an application and need maximum performance, then choose the JDBC OCI driver.
- If you are running in the Oracle database server using the Oracle 8.1.5 Java VM, then choose the JDBC Server driver.
## Requirements and Compatibilities for Oracle JDBC Drivers

Table 2–1 lists the compatibilities between versions of the Oracle database and the JDBC drivers.

### Table 2–1 JDBC Driver-Database Compatibility

<table>
<thead>
<tr>
<th>Database Version</th>
<th>Driver Version</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1.5</td>
<td>JDBC Thin Driver</td>
<td>Both client- and server-side drivers offer full object support when run against an 8.1.5 database.</td>
</tr>
<tr>
<td></td>
<td>JDBC OCI Driver</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JDBC Server Driver</td>
<td></td>
</tr>
<tr>
<td>8.1.4</td>
<td>JDBC Thin Driver</td>
<td>Both client- and server-side drivers offer full object support when run against an 8.1.4 database.</td>
</tr>
<tr>
<td></td>
<td>JDBC OCI Driver</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JDBC Server Driver</td>
<td></td>
</tr>
<tr>
<td>Note: the JDBC Server driver is not available for version 8.0.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.0.x</td>
<td>JDBC Thin Driver</td>
<td>The JDBC OCI and Thin drivers do not support objects when run against an 8.0.x database. This is because JDBC depends on PL/SQL functions that did not exist in 8.0.x.</td>
</tr>
<tr>
<td></td>
<td>JDBC OCI Driver</td>
<td></td>
</tr>
<tr>
<td>Note: the JDBC Server driver is not available for version 8.0.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.x</td>
<td>JDBC Thin Driver</td>
<td>The JDBC OCI and Thin drivers do not support objects when run against a 7.x database. This is because JDBC depends on PL/SQL functions that did not exist in 7.x</td>
</tr>
<tr>
<td></td>
<td>JDBC OCI Driver</td>
<td></td>
</tr>
<tr>
<td>Note: the JDBC Server driver is not available for version 7.x</td>
<td>The JDBC OCI driver does not support LOBs.</td>
<td></td>
</tr>
</tbody>
</table>
Verifying a JDBC Client Installation

This section has the following subsections:

- Check Installed Directories and Files
- Check the Environment Variables
- Make Sure You Can Compile and Run Java
- 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 Net8 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. The Oracle JDBC drivers are compatible with JDK versions 1.0.2 and 1.1.x. The Oracle JDBC drivers for version 8.1.5 do not support the JDK 1.2.

Directories for JDBC

Installing the Oracle Java server products creates, among other things, a jdbc directory under [ORACLE_HOME], containing these subdirectories and files:

- demo/samples: The samples directory contains sample programs, including examples of how to use SQL92 and Oracle SQL syntax, PL/SQL blocks, streams, and the Oracle JDBC type and performance extensions. The demo directory contains only the samples subdirectory.

- doc: The doc directory contains documentation about the JDBC drivers.

- lib: The lib directory contains .zip files with required Java classes: classes111.zip for JDK 1.1.1 and classes102.zip for JDK 1.0.2.
readme.txt: The readme.txt file contains up to the minute facts about the drivers that might not be in the 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.

Solaris and Windows NT Platforms

You must set the CLASSPATH for your installed JDBC OCI or Thin driver. Depending on whether you are using the JDK version 1.0.2 or version 1.1.1, you must set one of these values for the CLASSPATH:

- [Oracle Home]/jdbc/lib/classes102.zip
- [Oracle Home]/jdbc/lib/classes111.zip

OR
- [Oracle Home]/jdbc/lib/classes111.zip

**JDBC OCI Drivers:** If you are installing the JDBC OCI driver, you must also set the following value for the library path environment variable (this will be LD_LIBRARY_PATH on Solaris or PATH on Windows NT).
- [Oracle Home]/lib

On Solaris, this directory contains the shared object library libocijdbc8.so.

**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.
Determining the Version of the JDBC Driver

If at any time you need to 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:

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

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 illustrates several important functions:

- imports the necessary Java classes, including JDBC classes
Verifying a JDBC Client Installation

- 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.

```java
/*
 * 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.driver.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:oci8:8" + 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 "";
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. It includes the following topics:

- First Steps in JDBC
- Sample: Connecting, Querying, and Processing the Results
- Datatype Mappings
- Using Java Streams in JDBC
- Using Stored Procedures in JDBC Programs
- Error Messages and JDBC
- Server-Side Basics
- Application Basics versus Applet Basics
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. Importing Packages
2. Registering the JDBC Drivers
3. Opening a Connection to a Database
4. Creating a Statement Object
5. Executing a Query and Returning a Result Set Object
6. Processing the Result Set
7. Closing the Result Set and Statement Objects
8. Closing 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.

Importing Packages

Regardless of which Oracle JDBC driver you use, you must include the following import statements at the beginning of your program.

```java
import java.sql.*  // JDBC packages.
import java.math.*  // Java math packages; for example, these are required for the BigDecimal classes.
```

You will need to add the following Oracle packages to your program 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:
Registering 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.

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

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

```java
Class.forName ("oracle.jdbc.driver.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.

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

**Note:** If you are registering a Thin driver in an applet, you must enter a driver string that is different from the one used in these examples. For more information on registering a Thin driver for an applet, see "Coding Applets" on page 5-7.

Opening a Connection to a Database

You 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 which needs as input a userid, 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

```java
oracle.jdbc.driver.*
and oracle.sql.*
```

Add these packages if you use any Oracle-specific extensions to JDBC in your program. For more information on Oracle extensions, see Chapter 4, "Oracle Extensions".
familiar with this method, continue reading the "Understanding the Forms of
getConnection()" section below.

If you are already familiar with the get\nConnection() 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 3-6
- "Opening a Connection for the JDBC Thin Driver" on page 3-7

---

Note: The instructions in this section are specific to the client-side
drivers only. To find out how to open a database connection using
the server-side driver, see "Server-Side Basics" on page 3-26.

---

Understanding the Forms of get\nConnection()

The get\nConnection() method is an overloaded method that you declare by the
techniques described in these sections:

- "Specifying a Database URL, Userid, and Password" on page 3-4
- "Specifying a Database URL That Includes Userid and Password" on page 3-5
- "Specifying a Database URL and Properties Object" on page 3-6

---

Note: You do not have to specify the database name if there is a
default connection. For more information about default
connections, see "Connecting to the Database with the Server
Driver" on page 5-22.

---

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

- a Net8 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 Net8 Administrator’s Guide.

Specifying a Database URL, Userid, and Password

get\nConnection(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.

```java
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:

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

**OR**

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

For all JDBC drivers you can also specify the database with a Net8 `keyword-value` pair. The Net8 `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:

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

**OR**

```java
Connection conn =
    DriverManager.getConnection("jdbc:oracle:oci8:0 (description=(address=(host=myhost) (protocol=tcp) (port=1521)) (connect_data=(sid=orcl)))", "scott","tiger")
```

### Specifying a Database URL That Includes Userid and Password

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

where the URL is of the form:

```
jdbc:oracle:< drivertype >:<user>/<password>@<database>
```
First Steps in JDBC

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

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

### Specifying a Database URL and Properties Object

**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
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);
```

**Oracle Extensions to Connection Properties Object** Oracle has defined several extensions to the connection properties that Oracle JDBC drivers support. For more information on this form of the **getconnection()** method and the Oracle extensions to the **Properties** object, see "Oracle Extensions for Connection Properties" on page 4-109.

### 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 [ORACLE_HOME]\NETWORK\ADMIN. On UNIX systems, you can find it in /var/opt/oracle.

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:

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

Note that both the ":" and "@" characters are necessary.
For the JDBC OCI driver (as with the Thin driver), you can also specify the database with a Net8 "keyword-value" pair. This is less readable than a TNSNAMES entry but does not depend on the accuracy of the TNSNAMES.ORA file. The Net8 "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:

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

**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:

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

You can also specify the database with a Net8 "keyword-value" pair. This is less readable than the first version, but also works with the other JDBC drivers.

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

Executing a Query and Returning 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 an object of the JDBC `ResultSet` class.

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 that is named `EMP`:

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

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

---

**Note:** If you are writing a connection statement for an applet, you must enter a connect string that is different from the one used in these examples. For more information on connecting to a database with an applet, see "Coding Applets" on page 5-7.
Processing the Result Set

Once you execute your query, use the `next()` method of your `ResultSet` object to iterate through the results. This method loops 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 various `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:

```java
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.

Closing the Result Set and Statement Objects

You must explicitly close the `ResultSet` and `Statement` objects after you finish using them. This applies to all `ResultSet` and `Statement` objects you create when using 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:

```java
rset.close()
stmt.close();
```
When you close a Statement object that a given Connection object creates, the connection itself remains open.

Closing the Connection

You must close your connection to the database once you finish your work. Use the close() method of the Connection class to do this. For example, if your Connection object is conn, close the connection with this statement:

conn.close();

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.

```java
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.driver.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));
        }
    }
}
```
Datatype Mappings

If you want to adapt the code for the OCI driver, replace the Connection statement with the following:

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

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

---

**Note:** If you are creating code for an applet, the `getConnection()` and `registerDriver()` strings will be different. For more information, see "Coding Applets" on page 5-7.

---

Datatype Mappings

The Oracle JDBC drivers support the SQL datatypes required by JDBC 1.22. In addition, the Oracle JDBC drivers support the Oracle-specific ROWID datatype and user-defined types of the REF CURSOR category.

For reference, the following table shows the default mappings between JDBC datatypes, native Java datatypes, SQL datatypes, and the corresponding Java datatypes defined by Oracle extensions.

The **Standard JDBC Datatypes** column lists the datatypes supported by the JDBC 1.22 standard. All of these datatypes are defined in the `java.sql.Types` class.

The **Java Native Datatypes** column lists the datatypes defined by the Java language.

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

The **Oracle Extensions—Java Classes that Represent SQL Datatypes** 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 "Classes of the oracle.sql Package" on page 4-7 for more information on the `oracle.sql.*` package.
Datatype Mappings

For a list of all of the Java datatypes to which you can validly map a SQL datatype, see "Valid SQL-JDBC Datatype Mappings" on page 8-2.

Table 3–1  Mapping Between JDBC, Java Native, and Oracle Datatypes

<table>
<thead>
<tr>
<th>Standard JDBC Datatypes</th>
<th>Java Native Datatypes</th>
<th>SQL Datatypes</th>
<th>Oracle Extensions—Java Classes that Represent SQL Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.Types.CHAR</td>
<td>java.lang.String</td>
<td>CHAR</td>
<td>oracle.sql.CHAR</td>
</tr>
<tr>
<td>java.sql.Types.VARCHAR</td>
<td>java.lang.String</td>
<td>VARCHAR2</td>
<td>oracle.sql.CHAR</td>
</tr>
<tr>
<td>java.sql.Types.LONGVARCHAR</td>
<td>java.lang.String</td>
<td>LONG</td>
<td>oracle.sql.CHAR</td>
</tr>
<tr>
<td>java.sql.Types.NUMERIC</td>
<td>java.math.BigDecimal</td>
<td>NUMBER</td>
<td>oracle.sql.NUMBER</td>
</tr>
<tr>
<td>java.sql.Types.DATE</td>
<td>java.sql.Date</td>
<td>DATE</td>
<td>oracle.sql.DATE</td>
</tr>
<tr>
<td>java.sql.Types.TIME</td>
<td>java.sql.Time</td>
<td>DATE</td>
<td>oracle.sql.DATE</td>
</tr>
<tr>
<td>java.sql.Types.TIMESTAMP</td>
<td>java.sql.Timestamp</td>
<td>DATE</td>
<td>oracle.sql.DATE</td>
</tr>
<tr>
<td>java.sql.Types.TINYINT</td>
<td>byte</td>
<td>NUMBER</td>
<td>oracle.sql.NUMBER</td>
</tr>
<tr>
<td>java.sql.Types.SMALLINT</td>
<td>short</td>
<td>NUMBER</td>
<td>oracle.sql.NUMBER</td>
</tr>
<tr>
<td>java.sql.Types.INTEGER</td>
<td>int</td>
<td>NUMBER</td>
<td>oracle.sql.NUMBER</td>
</tr>
<tr>
<td>java.sql.Types.FLOAT</td>
<td>double</td>
<td>NUMBER</td>
<td>oracle.sql.NUMBER</td>
</tr>
<tr>
<td>java.sql.Types.DOUBLE</td>
<td>double</td>
<td>NUMBER</td>
<td>oracle.sql.NUMBER</td>
</tr>
<tr>
<td>java.sql.Types.BINARY</td>
<td>byte[]</td>
<td>NUMBER</td>
<td>oracle.sql.NUMBER</td>
</tr>
<tr>
<td>java.sql.Types.VARBINARY</td>
<td>byte[]</td>
<td>RAW</td>
<td>oracle.sql.RAW</td>
</tr>
<tr>
<td>java.sql.Types.LONGVARBINARY</td>
<td>byte[]</td>
<td>LONGRAW</td>
<td>oracle.sql.NUMBER</td>
</tr>
</tbody>
</table>

Oracle JDBC Extension Types

In addition, the following JDBC extensions for SQL datatypes (most of which comply with the JDBC 2.0 standard) are supported. They are not described until Chapter 4, "Oracle Extensions", but are summarized here for reference. Table 3–2 shows their mappings to Oracle datatypes.

The SQL Datatype column lists the SQL datatypes that exist in the database.
The **JDBC Extensions for SQL Datatypes** column lists the types into which Oracle datatypes should map according to the JDBC 2.0 standard. The class `oracle.jdbc.driver.OracleTypes.*` includes the definitions of Oracle-specific types that do not exist in the JDBC standard and is a superset of `oracle.sql.*`.

The **Oracle Extensions—Java Classes that Represent SQL Datatypes** 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. Refer to "Classes of the oracle.sql Package" on page 4-7 for more information on the `oracle.sql.*` package.

For a list of all of the Java datatypes to which you can validly map a SQL datatype, see "Valid SQL-JDBC Datatype Mappings" on page 8-2.

<table>
<thead>
<tr>
<th>SQL Datatype</th>
<th>JDBC Extensions for SQL Datatypes</th>
<th>Oracle Extensions—Java Classes that Represent SQL Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWID</td>
<td><code>oracle.jdbc.driver.OracleTypes.ROWID</code></td>
<td><code>oracle.sql.ROWID</code></td>
</tr>
<tr>
<td>user-defined types of the REF CURSOR category</td>
<td><code>oracle.jdbc.driver.OracleTypes.CURSOR</code></td>
<td><code>java.sql.ResultSet</code></td>
</tr>
<tr>
<td>BLOB</td>
<td><code>oracle.jdbc.driver.OracleTypes.BLOB</code></td>
<td><code>oracle.sql.BLOB</code></td>
</tr>
<tr>
<td>CLOB</td>
<td><code>oracle.jdbc.driver.OracleTypes.CLOB</code></td>
<td><code>oracle.sql.CLOB</code></td>
</tr>
<tr>
<td>BFILE</td>
<td><code>oracle.jdbc.driver.OracleTypes.BFILE</code></td>
<td><code>oracle.sql.BFILE</code></td>
</tr>
</tbody>
</table>
| Object Value  | `oracle.jdbc.driver.OracleTypes.STRUCT` | If there is no entry for the object value in the type map:  
|               |                                  | ■ oracle.sql.STRUCT  
|               |                                  | If there is an entry for the object value in the type map:  
|               |                                  | ■ customized Java class                                  |
| Object Reference | `oracle.jdbc.driver.OracleTypes.REF` | class that extends `oracle.sql.REF`                     |
| Collections (varrays and nested tables) | `oracle.jdbc.driver.OracleTypes.ARRAY` | `oracle.sql.ARRAY`                                     |

See Chapter 4, "Oracle Extensions", for more information on type mappings. In Chapter 4 you can also find more information on:

- packages `oracle.sql`, `oracle.jdbc.driver`, and `oracle.jdbc2`
- type extensions for the Oracle `ROWID` datatype and user-defined types of the `REF CURSOR` category
- how to use type maps with object values and collections
Using Java Streams in JDBC

This section has the following subsections:

■ 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: returns the RAW bytes of the data. This corresponds to the getBinaryStream() method.
■ ASCII stream: returns ASCII bytes in ISO-Latin-1 encoding. This corresponds to the getAsciiStream() method.
■ Unicode stream: returns 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 4, "Oracle Extensions".

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 "Using NLS" on page 5-2.

**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:

```
<table>
<thead>
<tr>
<th>LONG RAW</th>
<th>BinaryStream</th>
<th>ASCIIStream</th>
<th>UnicodeStream</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 21 22</td>
<td>20 21 22</td>
<td>49 52 49 53 49 54</td>
<td>0049 0052 0049 0053 0049 0054</td>
</tr>
<tr>
<td></td>
<td></td>
<td>which is also</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>'1' '4' '1' '5' '1' '6'</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>which is also:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'1' '4' '1' '5' '1' '6'</td>
<td></td>
</tr>
</tbody>
</table>
```

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 gibberish.

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 "Using NLS" on page 5-2.

Table 3–3 summarizes LONG and LONG RAW data conversions for each stream type.

### Table 3–3  LONG and LONG RAW Data Conversions

<table>
<thead>
<tr>
<th>Datatype</th>
<th>BinaryStream</th>
<th>AsciiStream</th>
<th>UnicodeStream</th>
</tr>
</thead>
<tbody>
<tr>
<td>LONG</td>
<td>bytes representing characters in Unicode UTF-8. The bytes can represent characters in US7ASCII or WE8ISO8859P1 if:</td>
<td>bytes representing characters in ISO-Latin-1 (WE8ISO8859P1) encoding</td>
<td>bytes representing characters in Unicode UCS-2 encoding</td>
</tr>
<tr>
<td></td>
<td>• the value of NLS_LANG on the client is US7ASCII or WE8ISO8859P1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• the database character set is US7ASCII or WE8ISO8859P1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONG RAW</td>
<td>as-is</td>
<td>ASCII representation of hexadecimal bytes</td>
<td>Unicode representation of hexadecimal bytes</td>
</tr>
</tbody>
</table>

### 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 of 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
-- 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:

```java
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 of the data in one call and stores it in a byte array. The previous code snippet can be rewritten as:

```java
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 the type `oracle.jdbc.driver.OracleStatement`.

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, data is returned by writing it to a byte array.

```java
//cast the statement stmt to an OracleStatement
oracle.jdbc.driver.OracleStatement ostmt = (oracle.jdbc.driver.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 `executeQuery()` or `next()`. The `getXXXStream()` entry points return a stream that reads data from this buffer.

**Note:** In version 8.1.5, the `setXXXStream()` methods are not available for CHAR, VARCHAR, and RAW datatypes.

### 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 usually not available until the stream has been read. This is because the database sends each row as a set of bytes representing the columns in the `SELECT` order: the data after a streaming column can be read only after the stream has been read.

For example, consider the following query:

```java
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:

\[\text{<a date><the characters of the long column><a number>}\]

When you call `rset.next()`, the JDBC driver stops reading the row data just before the first character of the `LONG` column. Then the driver uses `rset.getAsciiStream()` to read the characters of the `LONG` column directly out of the database connection as a Java stream. The driver reads the `NUMBER` data from the third column only after it reads the last byte of the data from the stream.

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 3-23.

**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:

```java
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 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 in the following sections:

- Use the Stream Data after You Access It
- Call the Stream Column in SELECT List Order

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 read and store 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. This is illustrated in the following example:

```java
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:

```java
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
}

Streaming and Row Prefetching

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

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 3-21. For information on how to avoid closing a stream and discarding its data by accident, see "Streaming Data Precautions" on page 3-22.

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. The JDBC drivers provide support for three types of LOBs: BLOBs (unstructured binary data), CLOBs (single-byte character data) and BFILEs (external files). The Oracle JDBC drivers support the streaming of CLOB, BLOB, and BFILE data.

LOBs behave differently from the other types of streaming data described in this chapter. The driver transfers LOB data between server and client as a Java 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.

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 4-48.

**Streaming BFILEs** An external file, or BFILE, is used to store a locator to a file that is 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 classes oracle.sql.BFILE. These classes provide 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 4-57.

### Using Stored Procedures in JDBC Programs

This section describes how the Oracle JDBC drivers support stored procedures and includes these subsections:

- 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 escape syntax. The following PL/SQL calls are all available from any Oracle JDBC driver:

// SQL92 Syntax
CallableStatement cs1 = conn.prepareCall
  ( "(call proc (?,?))" ) ;
CallableStatement cs2 = conn.prepareCall
  ( "(? = call func (?,?))" ) ;

// Oracle Syntax
CallableStatement cs3 = conn.prepareCall
  ( "begin proc (:1, :2); end;" ) ;
CallableStatement cs4 = conn.prepareCall
  ( "begin :1 := func(:2,:3); end;" ) ;
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 and concatenates a suffix to it:

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

Your invocation call in your JDBC program should look like:

```java
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 = proc.getString(1);
```

**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. See the Oracle8i Java Stored Procedures Developer's Guide for more information on using Java stored procedures.

**Error Messages and JDBC**

To handle exceptions, the Oracle JDBC drivers throw a `java.sql.SQLException()`. Two types of errors can be returned. The first type, Oracle database errors, are returned from the Oracle database itself and consist of an error number and a text message describing the error. These errors are documented in the publication Oracle8i Error Messages.

The second type of error is returned by the JDBC driver itself. These messages consist of a text message, but do not have an error number. These messages describe the error and identify the method that threw the error.

You can return errors with these methods:
getMessage(): returns the error message associated with the object that threw the exception

printStackTrace(): prints this object name and its stacktrace to the specified print stream

This example uses both getMessage() and printStackTrace() to return errors.

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

The text of all error messages has been internationalized. That is, they are available in all of the languages and character sets supported by Oracle. These error messages are listed in Appendix A, "JDBC Error Messages".

### Server-Side Basics

This section has the following subsections:

- **Session and Transaction Context**
- **Connecting to the Database**

The tutorial presented in "First Steps in JDBC" on page 3-2, describes connecting to and querying a database using the client-side driver. The following sections describe some of the basic differences if you run the tutorial using the server-side driver. For a complete discussion of the server-side driver, see "JDBC on the Server: the Server Driver" on page 5-22.

### Session and Transaction Context

The server-side 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 Driver" on page 5-23.

### Connecting to the Database

The Server driver uses a default connection to the database. You can connect to the database with either the DriverManager.getConnection() method or the
Oracle-specific API `defaultConnection()` method. For more information on connecting to the database with the server-side driver, see “Connecting to the Database with the Server Driver” on page 5-22.

Application Basics versus Applet Basics

This section has the following subsections:

- Application Basics
- Applet Basics

Application Basics

You can use either the Oracle JDBC Thin driver or the JDBC OCI driver to create an 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 run on the server by using the JDBC Server 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 Net8 and client libraries.

Applications and Encryption

For applications that use the Oracle OCI driver, you can encrypt data by using Net8 ANO (Advanced Networking Option). For more information on ANO, please refer to the `Net8 Administrator’s Guide`.

Applet Basics

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

Applets and Security

An applet cannot open network connections except 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, either:
- 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. If your browser supports JDK 1.1.x, then you can use signed applets. Signed applets can request socket connection privileges to other machines.

Both of these topics are described in greater detail in "Connecting an Applet to a Database" on page 5-9.

**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 5-14 for more information on configuring the firewall and on writing connect strings for the applet.

**Applets and Encryption**

Applets that use the JDBC Thin driver do not support data encryption.

**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 5-17.
This chapter describes Oracle extensions to standard JDBC, including the following topics:

- Introduction to Oracle Extensions
- Oracle JDBC Packages and Classes
- Data Access and Manipulation: Oracle Types vs. Java Types
- Working with LOBs
- Working with Oracle Object Types
- Working with Oracle Object References
- Working with Arrays
- Additional Oracle Extensions
- Oracle JDBC Notes and Limitations
Introduction to Oracle Extensions

Oracle’s implementation of JDBC supports versions 1.0.2 and 1.1.x of the Sun Microsystems JDK and complies with JDBC 1.22, included with these JDK versions. This chapter describes Oracle extensions to JDBC 1.22, organized into two categories:

- type extensions to comply with a subset of JDBC 2.0, which is a part of JDK 1.2
- additional Oracle-specific type extensions and performance extensions

This section describes how Oracle JDBC supports these extensions, including the Java packages created and the datatype support issues that must be considered.

Note: The JDBC OCI, Thin, and Server drivers support the same functionality, and all of the Oracle extensions.

Packages Oracle release 8.1.5 does not support JDK 1.2. The JDBC 2.0 interfaces are part of the java.sql package that is included with the JDK 1.2. Therefore, to support JDBC 2.0 types, as well as additional Oracle extensions, the Oracle JDBC distribution includes the following Java packages:

- oracle.jdbc2 (a subset of the standard JDBC 2.0 interfaces)
- oracle.sql (classes to support all Oracle type extensions)
- oracle.jdbc.driver (classes to support database access and updates in Oracle type formats)

"Oracle JDBC Packages and Classes" on page 4-6 further describes these packages and their classes.

Oracle Datatype Support 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 of 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 of the necessary methods to convert data to appropriate Java formats.

For a more detailed description of these general issues, see "Classes of the `oracle.sql` Package" on page 4-7.

Specific information relating to particular `oracle.sql.*` datatype classes is discussed in the sections "Working with LOBs" on page 4-45 and "Additional Type Extensions" on page 4-111.

**Oracle Object Support** Perhaps the most noteworthy Oracle8 type is Oracle objects. Oracle8 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 Java format or in `oracle.sql.*` format)
- how to convert attribute data between SQL and Java formats
- how to access data

To manually create Java classes to correspond to your Oracle objects, Oracle recommends that you use the Oracle8i JPublisher utility to create the classes. To do this, you must define attributes according to how you want to store the data. JPublisher handles this task seamlessly with command-line options.

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.

Each Java class created to correspond to an Oracle object datatype must implement one of two supported interfaces: the JDBC-standard `SQLData` interface or the `OracleCustomDatum` interface. Each of these interfaces specifies methods to convert data between SQL and Java. Currently, JPublisher supports only the `CustomDatum` interface.
JPublisher automatically defines `get` methods of the Java classes, which retrieve data into your Java application. For more information on the JPublisher utility, see the Oracle8i JPublisher User’s Guide.

"Working with Oracle Object Types" on page 4-62 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 the `schema_name` and the `sql_type_name` is 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.

- The JDBC driver assumes that schema names do not contain dots (".").
- The JDBC driver does not allow double quotes (" ") as part of the schema name or the type name.
Oracle JDBC Packages and Classes

This section discusses the Java packages that support the Oracle JDBC extensions and the key classes that are included in these packages. This section has the following subsections:

- Classes of the oracle.jdbc2 Package
- Classes of the oracle.sql Package
- Classes of the oracle.jdbc.driver Package

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

Classes of the oracle.jdbc2 Package

The `oracle.jdbc2` package contains the Oracle implementation of the standard JDBC 2.0 interfaces. The JDBC 2.0 interfaces are part of the `java.sql` package included in the JDK 1.2. However, since the drivers do not currently support JDK 1.2, these interfaces have been made available to the Oracle 1.0.2 and 1.1.x drivers as the `oracle.jdbc2` package. This package contains the JDBC 2.0 features of the JDK 1.2 `java.sql` package that the Oracle drivers support.

The following interfaces are implemented by `oracle.sql.*` type classes for JDBC 2.0-compliant Oracle type extensions. These interfaces are equivalent to the interfaces published by Sun Microsystems; the `oracle.jdbc2` versions add no new features.

- `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, Oracle includes the following standard JDBC 2.0 interfaces for users employing the JDBC-standard `SQLData` interface to create Java classes that map to Oracle objects:

- `oracle.jdbc2.SQLData` implemented by classes that map to Oracle objects; users must provide this implementation
- `oracle.jdbc2.SQLInput` implemented by classes that read object data; Oracle provides a `SQLInput` class that the JDBC drivers use
oracle.jdbc2.SQLOutput 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 features you can use to support Oracle objects in Java. The other feature is the Oracle CustomDatum interface, contained in the oracle.sql package. See “Understanding the SQLData Interface” on page 4-69 for more information about SQLData, SQLInput, and SQLOutput.

---

**Note:** Oracle recommends using the CustomDatum interface instead of the SQLData interface. CustomDatum works more easily in conjunction with other features of the Oracle Java product offerings, such as the JPublisher utility (which can automatically generate CustomDatum classes corresponding to Oracle objects) and SQLJ.

---

**Classes of the oracle.sql Package**

The oracle.sql package supports direct access to data in SQL format and consists primarily of classes that map to the Oracle SQL datatypes.

These classes provide Java mappings for the Oracle SQL types and are wrapper classes 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 datatype classes extends oracle.sql.Datum, a superclass that encapsulates functionality common to all of the datatypes. Some of the classes are for JDBC 2.0-compliant datatypes. These classes, as Table 4–1 indicates, implement standard JDBC 2.0 interfaces in the oracle.jdbc2 package, as well as extending oracle.sql.Datum.

**Table 4–1** lists the oracle.sql datatype classes and their corresponding Oracle SQL types.

<table>
<thead>
<tr>
<th>Java Class</th>
<th>Oracle SQL Type (and Description) and Interface Implemented if for JDBC 2.0</th>
</tr>
</thead>
</table>
| oracle.sql.STRUCT | STRUCT (objects)  
                        | JDBC 2.0, implements oracle.jdbc2.Struct                                    |
Table 4–1  Oracle Datatype Classes (Cont.)

<table>
<thead>
<tr>
<th>Java Class</th>
<th>Oracle SQL Type (and Description) and Interface Implemented if for JDBC 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>oracle.sql.REF</td>
<td>REF (object references) JDBC 2.0, implements oracle.jdbc2.Ref</td>
</tr>
<tr>
<td>oracle.sql.ARRAY</td>
<td>varray or nested table (collections) JDBC 2.0, implements oracle.jdbc2.Array</td>
</tr>
<tr>
<td>oracle.sql.BLOB</td>
<td>BLOB (large binary objects) JDBC 2.0, implements oracle.jdbc2.Blob</td>
</tr>
<tr>
<td>oracle.sql.CLOB</td>
<td>CLOB (large character objects) JDBC 2.0, implements oracle.jdbc2.Clob</td>
</tr>
<tr>
<td>oracle.sql.BFILE</td>
<td>BFILE (external files)</td>
</tr>
<tr>
<td>oracle.sql.CHAR</td>
<td>CHAR, VARCHAR2</td>
</tr>
<tr>
<td>oracle.sql.DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>oracle.sql.NUMBER</td>
<td>NUMBER</td>
</tr>
<tr>
<td>oracle.sql.RAW</td>
<td>RAW</td>
</tr>
<tr>
<td>oracle.sql.ROWID</td>
<td>ROWID (row identifiers)</td>
</tr>
</tbody>
</table>

The following sections describe each class listed in Table 4–1. Additional details about use of the Oracle extended types (STRUCT, REF, ARRAY, BLOB, CLOB, BFILE, and ROWID) are described in "Working with LOBs" on page 4-45, "Working with Oracle Object References" on page 4-83, "Working with Arrays" on page 4-87, and "Additional Type Extensions" on page 4-111.
In addition to the datatype classes, the oracle.sql package includes these support classes and interfaces:

- **oracle.sql.ArrayDescriptor class**: used in constructing oracle.sql.ARRAY objects; describes the SQL type of the array. See "Class oracle.sql.ARRAY" on page 4-14 for more information.
- **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 "Class oracle.sql.STRUCT" on page 4-10 for more information.
- **oracle.sql.CharacterSet and oracle.sql.CharacterSetFactory classes**: used in constructing character set objects, which in turn are used in constructing oracle.sql.CHAR objects. See "Class oracle.sql.CHAR" on page 4-19 for more information.
- **oracle.sql.CustomDatum and oracle.sql.CustomDatumFactory interfaces**: used in Java classes implementing the Oracle CustomDatum scenario of Oracle object support. (The other possible scenario is the JDBC-standard SQLData implementation. See "Understanding the CustomDatum Interface" on page 4-75 for more information on CustomDatum.)
Refer to the Javadoc for additional information about these classes. The rest of this section further describes the oracle.sql.* classes.

**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
- 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`.

- a `stringValue()` or `intValue()` method, where appropriate, to convert the SQL data to a String or an int
- additional conversion, get, and set 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 Javadoc for additional information about these classes.

**Class oracle.sql.STRUCT**

For any given Oracle object type, if you do not specify a mapping to a Java class in your connection’s type map, 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 `oracle.jdbc2.Struct` class and extends `oracle.sql.Datum`.

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 and contains a "values" array of `oracle.sql.Datum` objects holding the attribute values in SQL format. The `STRUCT` object also contains the SQL type name of the Oracle object.

In most cases you will probably want to create a custom Java type definition class to map to your Oracle object, although using the `STRUCT` class may suffice in some
cases (see "Using STRUCT Objects" on page 4-63). The attributes of a STRUCT can be materialized as java.lang.Object[] objects if you use the getAttributes() method, or as oracle.sql.Datum[] objects if you use the getOracleAttributes() method. The oracle.sql.* format gives you the same advantages as using oracle.sql.* datatyp classes in general:

- The STRUCT class completely preserves data, because it maintains the data in SQL format. This is useful if you want to manipulate 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, would actually contain data associated with the relevant oracle.sql.* type appropriate for the given attribute, such as oracle.sql.CHAR in the case of CHAR data. You can cast an element as the appropriate oracle.sql.* type as desired.
- The JDBC driver materializes nested objects in the values array of a STRUCT object as instances of STRUCT themselves.
- Refer to the Javadoc for more information about particular features and methods of the oracle.sql.STRUCT class.

---

In some cases you might want to manually create a STRUCT object to pass it to a prepared statement or callable statement. To do this, you must also create a StructDescriptor object. For more information on creating a STRUCT object, see "Creating STRUCT Objects and Descriptors" on page 4-13.

The STRUCT class includes the following methods:

- getAttributes(): retrieves the values from the values array, using the type map (if one has been defined) to determine which Java classes to use in materializing the data. Conceptually, getAttributes() returns a Java array containing the attribute values. The types of the attribute values are those that a call to getObject() on the same underlying types will return. That is, they are the "default" JDBC types for the corresponding underlying types.

For example, assume that you have defined a SQL type PERSON with a name attribute of type CHAR and an age attribute of type NUMBER. If you use getAttributes() to get the object attributes of PERSON, then it will return the name as a Java String type and the age as a Java BigDecimal type.
If you are calling `getAttributes()` on a nested object, then you can optionally specify a type map (`java.util.Map` object) if you do not want to use your connection’s default type map.

- `getOracleAttributes()`: retrieves the values of the values array as `oracle.sql.*` objects
- `getSQLTypeName()`: returns the fully qualified type name (`schema.sql_type_name`) of the Oracle object that this `STRUCT` represents
- `getDescriptor()`: returns the `StructDescriptor` object for this `STRUCT` object (see “Creating `STRUCT` Objects and Descriptors” on page 4-13 for information about the `StructDescriptor` class)
- `getConnection()`: returns the current connection
- `getDescriptor()`: returns the `OracleType` that identifies the Oracle object type
- `getMap()`: returns the current type map
- `isConvertibleTo(Class)`: determines if a datum object can be converted to a particular class
- `makeJdbcArray(int)`: returns a JDBC array representation of the datum
- `setDatumArray(Datum[])`: sets the `Datum` array.
- `setDescriptor(StructDescriptor)`: sets the descriptor
- `stringValue()`: converts to a String representation of the datum object
- `toBytes()`: packs the bytes representing the attributes into the format that is actually used in the database
- `toClass(Class)`: applies the normal algorithms for converting a SQL structured type to a specific Java class
- `toJdbc()`: consults the current map to determine what class to convert to, and then uses `toClass()`
- `toJdbc(Dictionary)`: consults the map to determine what class to convert to, and then uses `toClass()`
- `toSTRUCT(Object, OracleConnection)`: returns the corresponding `STRUCT` object from the input Java object
Creating STRUCT Objects and Descriptors  To create an oracle.sql.STRUCT object, a STRUCT descriptor must first exist for the given Oracle object type. This descriptor is an object of the oracle.sql.StructDescriptor class.

A StructDescriptor describes a type of SQL structured object (Oracle object). Only one StructDescriptor is necessary for each Oracle object type.

The driver caches STRUCT descriptor objects to avoid recreating them if the type has already been encountered. The Oracle JDBC extensions provide a static createDescriptor() method that will either construct a new StructDescriptor object or return an existing one.

To create a StructDescriptor object, pass in a Java string parameter with the SQL type name of the Oracle object type and a connection object to the StructDescriptor.createDescriptor() method:

```java
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.

You can also call the StructDescriptor object if you need to create a new STRUCT object. To construct a new StructDescriptor object, pass in a Java string parameter with the SQL type name of the Oracle object type and your connection object:

```java
StructDescriptor structdesc = new StructDescriptor(sql_type_name, connection);
```

To construct a STRUCT object, pass in the StructDescriptor, your connection object, and an array of Java objects containing the attributes you want the STRUCT to contain.

```java
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 get Methods  A STRUCT descriptor can be referred to as a "type object." This means that it contains information about the type code and type name of the object type 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 `getName()` method to return the fully qualified SQL type name of the Oracle object (that is, in `schema.sql_type_name` format. For example, `CORPORATE.EMPLOYEE`)

**Embedded Objects** 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 else using default mapping.

**Class oracle.sql.REF**
The **oracle.sql.REF** class is the generic class that supports Oracle object references. This class, as with all of the `oracle.sql.*` datatype classes, is a subclass of `oracle.sql.Datum`. It implements the standard JDBC 2.0 `oracle.jdbc2.Ref` interface.

Selecting a `REF` retrieves only a pointer to an object; it does not materialize the object. However, there are methods to accomplish this.

The **oracle.sql.REF** class includes the following methods:

- `getValue()`: retrieves object attributes (using your type map as necessary)
- `setValue()`: sets object attributes (using your type map as necessary)
- `getBaseTypeName()`: retrieves the fully-qualified SQL structured type name of the referenced item

The `setREF()` and `setRef()` methods of the `OraclePreparedStatement` and `OracleCallableStatement` classes support passing a `REF` object as an input parameter to a prepared statement. Similarly, the `getREF()` and `getRef()` methods of the `OracleCallableStatement` and `OracleResultSet` support passing a `REF` object as an output parameter.

You cannot create `REF` objects using JDBC.

For more information on how to use `REF` objects, see "Working with Oracle Object References" on page 4-83.

**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 (as do all of the oracle.sql.* classes) and implements oracle.jdbc2.Array, a standard JDBC 2.0 array interface.

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, which "Creating ARRAY Objects and Descriptors" on page 4-15 describes.

The ARRAY class includes the following methods:

- `getArray()`: retrieves the contents of the array in "default" JDBC types. If it retrieves an array of objects, then `getArray()` uses the type map to determine the types
- `getOracleArray()`: identical to `getArray()`, but retrieves the elements in oracle.sql.* format
- `getArrayDescriptor()`: returns the ArrayDescriptor object that pertains to this array (see "Creating ARRAY Objects and Descriptors" on page 4-15 for information about the ArrayDescriptor class)
- `getBaseType()`: returns the SQL type code for the array elements (see "Class oracle.jdbc.driver.OracleTypes" on page 4-28 for information about type codes)
- `getSQLTypeName()`: returns the SQL type name of the array elements
- `getBaseTypeName()`: for named types (such as Oracle objects), returns the particular type name (for example, EMPLOYEE)
- `getResultSet()`: materializes an array as a result set

Creating ARRAY Objects and Descriptors The `setARRAY()` method of the OraclePreparedStatement or OracleCallableStatement class supports passing an array as an input parameter to a prepared statement. You must first construct an array descriptor, which is an oracle.sql.ArrayDescriptor object, and then you must construct the oracle.sql.ARRAY object for the array you want to pass.

An ArrayDescriptor object describes the SQL type of an array; however, you need only one array descriptor for any one SQL type. 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 arrays", that is, an array given a SQL type name. For example, when you create an array with the CREATE TYPE statement:
CREATE TYPE num_varray AS varray(22) OF NUMBER(5,2);

the SQL type name for the collection type is num_varray.

---

**Note:** The name of the collection type has nothing to do with 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 type name of the collection type is array_of_persons. The SQL type name of the elements of the collection is person.

---

To construct an ArrayDescriptor object, pass 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.

To construct an ARRAY object, 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 of objects. The two possibilities for the contents of elements are:

- an array of Java primitives. For example, int[].
- an array of Java objects. (For example, xxx[] where xxx represents the name of a Java object type.) For example, Integer[].
Notes:

- 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.
- Refer to the Javadoc for more information about the features of the `ARRAY` and `ArrayDescriptor` classes.

Using `ArrayDescriptor` get Methods An array descriptor can be referred to as a type object, meaning it has information about the array’s SQL type name, the type code of the array’s elements and, if the array is a `STRUCT`, the type name of the elements. The array descriptor also contains the information on how to convert to and from the given type. You need only one array 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 type code and type name:

- `getBaseType()`: returns the integer type code associated with this array descriptor (according to integer constants defined in the `OracleTypes` class, which "Classes of the oracle.jdbc.driver Package" on page 4-22 describes)
- `getBaseName()`: returns a string with the type name associated with this array element if it is a `STRUCT`, `REF` or collection.

Note: The elements of an array cannot be of type `ARRAY`. Collections cannot have elements of type `collection`. But Oracle objects and `STRUCT`S can have attributes of Java type `ARRAY` (SQL type `collection`).

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

BLOBs, CLOBs, and BFILEs, all referred to as LOBs, are for data items that are too large to store directly in the database table. Instead, the database table stores a locator that points to the location of the actual data.

The `oracle.sql` package supports LOBs in several ways:

- BLOBs point to large unstructured binary data items and are supported by the `oracle.sql.BLOB` class.
Oracle JDBC Packages and Classes

- **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. This is described in "Working with LOBs" on page 4-45.

---

**Note:** The `oracle.sql.CLOB` class supports all character sets that the Oracle data server supports for CLOB types.

---

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

- `getBinaryOutputStream()`: returns the BLOB data
- `getBinaryStream()`: returns the BLOB designated by this Blob instance as a stream of bytes
- `getBytes()`: reads from the BLOB data, starting at a specified point, into a supplied buffer
- `length()`: returns the length of the BLOB in bytes
- `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

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

- `getAsciiOutputStream()`: writes CLOB data from an ASCII stream
- `getAsciiStream()`: returns the CLOB value designated by the Clob object as a stream of Ascii bytes
- `getCharacterOutputStream()`: writes CLOB data from a Unicode stream
- `getCharacterStream()`: returns the CLOB data as a stream of Unicode characters
- `getChars()`: retrieves characters from a specified point in the CLOB data into a character array
length(): returns the length of the CLOB in characters

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

The oracle.sql.BFILE class includes the following methods:

openFile(): opens the external file

closeFile(): closes the external file

getBinaryStream(): returns the contents of the external file as a stream of bytes

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

---

**Note:** You cannot write to a BFILE; you can only read from it.

---

Class oracle.sql.CHAR

The CHAR class has special functionality for NLS conversion of character data. A key attribute of the CHAR class, and a parameter always passed in when a CHAR object is constructed, is the NLS character set used in presenting the character data. Without the character set being known, the bytes of data in the CHAR object are meaningless.

CHAR objects that the driver constructs and returns can be in the database character set, UTF-8, or ISO-Latin-1 (WE8ISO8859P1). CHAR objects which are Oracle8 objects, are returned in the database character set.
JDBC constructs and populates CHAR objects once character data has been read from the database. Additionally, you might want to construct a CHAR object yourself (to pass in to a prepared statement, for example).

When you construct a CHAR object, you must provide character set information to the CHAR object by way of an instance of the oracle.sql.CharacterSet class. Each instance of the CharacterSet class represents one of the NLS 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 Oracle8i National Language Support Guide.

If you use a CHAR object based on a character set that Oracle does not support, then the JDBC driver will not be able to perform character set conversions with it. For example, you will not be able to use the CHAR object in an OraclePreparedStatement.setOracleObject() call.

Follow these general steps to construct a CHAR object:

1. Create a CharacterSet instance by calling the static CharacterSet.make() method. This method is a factory for the character set class. It takes as input an integer OracleId, which corresponds to a character set that Oracle supports. For example:

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

   Each character set that Oracle supports has a unique predefined OracleId. If you enter an invalid OracleId, an exception will not be thrown. Instead, when you try to use the character set, you will receive unpredictable results. For more information on character sets and character set IDs, see the Oracle8i National Language Support Guide.

2. Construct a CHAR object. Pass to the constructor a string (or the bytes that represent the string) and the CharacterSet object that indicates how to interpret the bytes based on the character set. For example:

   ```java
   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 CharSet object before being placed into the CHAR object.

Refer to the CHAR class Javadoc for more information.

---

**Notes:**

- The CharSet object cannot be null.
- The CharSet class is an abstract class, therefore it has no constructor. The only way to create instances is through use of the make() method.
- The server recognizes the special value CharSet.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 CharSet class.

---

The CHAR class provides these 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 the character set is not recognized (that is, if you entered an invalid OracleID), then getString() throws a SQLException.
- **toString():** identical to getString(), but if the character set is not recognized (that is, if you entered an invalid OracleID), then toString() 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 character set of this CHAR object. This default character varies from character set to character set, but is often a question mark.

The server (database) and the client (or application running on the client) can use different character sets. When you use the methods of this 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 Oracle’s National Language Support (NLS). For more information on how the JDBC drivers convert between character sets, see "Using NLS" on page 5-2. For more information on NLS, see the Oracle8i National Language Support Guide.
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. These classes provide conversions to and from their corresponding JDBC Java types. For more information, see the Javadoc.

**Class `oracle.sql.ROWID`**

This class supports Oracle `ROWID`s, 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 `ROWID`s; 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 "Additional Oracle Extensions" on page 4-97.

**Classes of the `oracle.jdbc.driver` Package**

The `oracle.jdbc.driver` package includes classes that add extended features to enable data access in `oracle.sql` format. In addition, these classes provide Oracle-specific extensions to allow access to raw SQL format data by using `oracle.sql.*` objects.

Table 4–2 lists key classes for connections, statements, and result sets in this package.

<table>
<thead>
<tr>
<th>Class</th>
<th>Key Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>OracleDriver</td>
<td>implements <code>java.sql.Driver</code></td>
</tr>
<tr>
<td>OracleConnection</td>
<td>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>)</td>
</tr>
</tbody>
</table>
The `oracle.jdbc.driver` package additionally includes:

- stream classes
- the `OracleTypes` class

The stream classes extend standard Java stream classes and read and write Oracle LOB, LONG, and LONG RAW data.

`OracleTypes` defines integer constants, which identify SQL types. For standard types, it uses the same values as the standard `java.sql.Types`. In addition, it adds constants for Oracle extended types.

The remainder of this section describes the classes of the `oracle.jdbc.driver` package. For more information about using these classes to access Oracle type extensions, see "Data Access and Manipulation: Oracle Types vs. Java Types" on page 4-32.

### Table 4-2  Connection, Statement, and Result Set Classes (Cont.)

<table>
<thead>
<tr>
<th>Class</th>
<th>Key Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>OracleStatement</td>
<td>methods to set Oracle performance extensions for individual statement; superclass of OraclePreparedStatement and OracleCallableStatement (implements java.sql.Statement)</td>
</tr>
<tr>
<td>OraclePreparedStatement</td>
<td>set methods to bind oracle.sql.* types into a prepared statement (implements java.sql.PreparedStatement; extends OracleStatement)</td>
</tr>
<tr>
<td>OracleCallableStatement</td>
<td>get methods to retrieve data in oracle.sql format; set methods to bind oracle.sql.* types into a callable statement (inherited from OraclePreparedStatement) (implements java.sql.CallableStatement; extends PreparedStatement)</td>
</tr>
<tr>
<td>OracleResultSet</td>
<td>get methods to retrieve data in oracle.sql format (implements java.sql.ResultSet)</td>
</tr>
<tr>
<td>OracleResultSetMetaData</td>
<td>methods to get information about Oracle result sets (implements java.sql.ResultSetMetaData)</td>
</tr>
</tbody>
</table>
Class oracle.jdbc.driver.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.

Class oracle.jdbc.driver.OracleConnection

This class 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.

"Performance Extensions" on page 4-97 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
- getTransactionIsolation(): gets this connection’s current isolation mode
- setTransactionIsolation(): changes the transaction isolation level using one of the TRANSACTION_\* values

These oracle.jdbc.driver.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

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)

Class oracle.jdbc.driver.OracleStatement
This class extends standard JDBC statement functionality and is the superclass 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 class that sets these on a connection-wide basis.

"Performance Extensions" on page 4-97 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.driver.OracleStatement methods are Oracle-defined extensions:

defineColumnType(): defines the type you will use to retrieve data from a particular database table column

ggetRowPrefetch(): retrieves the row-prefetch value for this statement

setRowPrefetch(): sets the row-prefetch value for this statement

Class oracle.jdbc.driver.OraclePreparedStatement
This class extends standard JDBC prepared statement functionality, is a subclass of the OracleStatement class, and is the superclass of the
OracleCallableStatement class. Extended functionality consists of set methods for binding oracle.sql.* types and objects into prepared statements, and methods to support Oracle performance extensions on a statement-by-statement basis.

"Performance Extensions" on page 4-97 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()` a generic set method for binding oracle.sql.* data into a prepared statement as an oracle.sql.Datum object
- `setXXX()` set methods, such as `setBLOB()`, for binding specific oracle.sql.* types into prepared statements. For more information on all of the `setXXX()` methods available for oracle.sql.* types, see the Javadoc.
- `setCustomDatum()` binds a CustomDatum 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.
- `close()` closes the current statement

Class oracle.jdbc.driver.OracleCallableStatement

This class extends standard JDBC callable statement functionality and is a subclass of the OracleStatement and OraclePreparedStatement classes. Extended functionality includes set methods for binding structured objects and oracle.sql.* objects into prepared statements, and get methods for retrieving data into oracle.sql.* objects.

Key methods include:

- `getOracleObject()` a generic get method for retrieving data into an oracle.sql.Datum object. It can be cast to the specific oracle.sql.* type as necessary.
- `getXXX()` get methods, such as `getCLOB()`, for retrieving data into specific oracle.sql.* objects. For more information on all of the `getXXX()` methods available for oracle.sql.* types, see the Javadoc.
- `setOracleObject()`: a generic set method for binding `oracle.sql.*` data into a callable statement as an `oracle.sql.Datum` object

- `setXXX()`: set methods inherited from `OraclePreparedStatement`, such as `setBLOB()`, for binding specific `oracle.sql.*` objects into callable statements. For more information on all of the `setXXX()` methods available for `oracle.sql.*` types, see the Javadoc.

- `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.

- `registerOutParameter()`: registers the SQL type code 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.driver.OracleTypes`). This is an overloaded method. There is a version of this method that you use for named types only; that is, when the SQL type code is `OracleTypes.REF`, `STRUCT`, or `ARRAY`. In this case, in addition to a parameter index and SQL type, the method also takes a `String` SQL type name (the name of the Oracle object type in the database, such as `EMPLOYEE`).

- `close()`: closes the current result set, if any, and the current statement

### Class `oracle.jdbc.driver.OracleResultSet`

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

Key methods include:

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

- `getXXX()`: get methods, such as `getCLOB()`, for retrieving data into `oracle.sql.*` objects

- `next()`: advances to the next row of the result set
Class oracle.jdbc.driver.OracleResultSetMetaData

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

Key methods include the following:

- getColumnCount(): returns the number of columns in an Oracle result set
- getColumnName(): returns the name of a specified column in an Oracle result set
- getColumnType(): 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.
- getColumnTypeName(): returns the SQL type name of the data stored in the column. 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 REF points.
- getTableName(): returns the name of the table from which an Oracle result set column was selected

Oracle Stream Classes

Oracle uses many stream classes that extend standard Java stream classes to provide special functionality, such as writing directly to an Oracle database. The JDBC drivers use these classes which are in the oracle.jdbc.driver package but does not intend them for use by Java applications programmers. For more information on Java streams, see “Using Java Streams in JDBC” on page 3-14.

Class oracle.jdbc.driver.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.driver.OracleTypes class contains a copy of the standard Java java.sql.Types class and contains these additional Oracle type extensions:

- OracleTypes.STRUCT
- OracleTypes.REF
- OracleTypes.ARRAY
- OracleTypes.BLOB
- OracleTypes.CLOB
Oracle JDBC Packages and Classes

- OracleTypes.BFILE
- OracleTypes.ROWID

As in java.sql.Types, all of 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 SQL types in the OracleTypes class identify the SQL type of the output parameters in the registerOutParameter() method of the java.sql.CallableStatement and oracle.jdbc.driver.OracleCallableStatement classes.

These are the forms that registerOutParameter() can take for CallableStatement and OracleCallableStatement:

```java
CallableStatement.registerOutParameter(int index, int sqlType)

CallableStatement.registerOutParameter(int index, int sqlType, int scale)

OracleCallableStatement.registerOutParameter(int index, int sqlType, String sql_name)
```

In these prototypes, index represents the parameter index, sqlType represents the SQL datatype (one of the OracleTypes, in this case), sql_name represents the name given to the datatype (that is, the "named type"), and scale represents the number of digits to the right of the decimal point when sqlType is a NUMERIC or DECIMAL datatype.

Any output parameter datatype except STRUCT, ARRAY, or REF can use the two forms of CallableStatement.registerOutParameter().

The OracleCallableStatement form of registerOutParameter() can be used only when the output parameter is of type STRUCT, ARRAY, or REF and requires you to provide the name of the named type.

The following example uses a CallableStatement to call a procedure named procout, which returns a CHAR datatype. Note the use of the OracleTypes.CHAR SQL name in the registerOutParameter() method.

```java
CallableStatement procout = conn.prepareCall("BEGIN procout (?); END;");
    procout.registerOutParameter (1, OracleTypes.CHAR);
    procout.execute ();
    System.out.println ("Out argument is: " + procout.getString (1));
```
The next example uses a CallableStatement to call procout, which returns a STRUCT datatype. The form of registerOutParameter() requires you to specify the name of the SQL type, OracleTypes.STRUCT, as well as the SQL type name (that is, the name of the named type) 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 into the default STRUCT datatype, the statement object procout is cast to an OracleCallableStatement and the getSTRUCT() is applied.

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

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

**OracleTypes and the setNull() Method** The SQL types in the OracleTypes class identify the object, which the setNull() method sets to NULL. The setNull() method can be found in the java.sql.PreparedStatement and oracle.jdbc.driver.OraclePreparedStatement classes.

These are the forms that setNull() can take for PreparedStatement and OraclePreparedStatement classes:

- `PreparedStatement.setNull(int index, int sqlType)`
- `OraclePreparedStatement.setNull(int index, int sqlType, String sql_name)`

In these prototypes, `index` represents the parameter index, `sqlType` represents the SQL datatype (one of the OracleTypes, in this case), and `sql_name` represents the name given to the datatype (that is, the name of the "named type"). If you enter an invalid sqlType, a "Parameter Type Conflict" error is thrown.

You can use the PreparedStatement form of setNull() to set to NULL the value of an object of any datatype, except STRUCT, ARRAY, or REF.

You can use the OraclePreparedStatement form of setNull() only when you set to NULL the value of an object of datatype STRUCT, ARRAY, or REF.

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 that is set to NULL.

```java
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. Note that an OraclePreparedStatement is required to set a STRUCT object to NULL. Thus, the prepared statement pstmt must be cast to OraclePreparedStatement.

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

((OraclePreparedStatement)pstmt).setNull(1, OracleTypes.STRUCT, "EMPLOYEE");
pstmt.execute();
Data Access and Manipulation: Oracle Types vs. Java Types

This section contains the following subsections:

- Data Conversion Considerations
- Using Result Set and Statement Extensions
- Comparing get and set Methods for oracle.sql.* Format with Java Format
- Using Result Set Meta Data Extensions

This section describes data access in oracle.sql.* formats as opposed to Java formats. As discussed in the introduction to this 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() (where XXX corresponds to the types in the oracle.sql package) methods of these classes. Refer to the Javadoc for additional information about these classes and methods.

Data Conversion Considerations

When JDBC programs retrieve SQL data into Java variables, the SQL data is converted to the Java datatypes of those variables. The Java datatypes can be represented as members of the oracle.sql package instead of as members of the java.lang or java.sql.Types packages. 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; 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 of the database, then you will probably want to represent the data as a member of java.sql.Types.* or java.lang.*. Similarly, if you are using a parser that expects the data to be in Java
Data Access and Manipulation: Oracle Types vs. Java Types

Converting SQL NULL Data
Java represents a SQL NULL datum by the Java value null. Java datatypes fall into two categories: the fixed set of scalar types (such as byte, int, float) and object types (such as objects and arrays). The Java scalar 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 scalar 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.

Using 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. The object is unchanged. The type by which the Java compiler will identify the object is changed.

When you execute a SELECT statement in a Java application using a standard JDBC Statement object, Oracle's JDBC drivers return a java.sql.ResultSet object. You can use this standard ResultSet object if all you need are standard JDBC ResultSet methods, but to use Oracle extensions you must cast the result set to an OracleResultSet object.

For example, assuming you have a standard Statement object stmt, do the following if you want to use only standard JDBC ResultSet methods:

```java
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. The object is unchanged. The type by which
the Java compiler identifies the object is changed.

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. The object is unchanged. The type by
which the Java compiler identifies the object is changed.

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: “Comparing get and set
Methods for oracle.sql.* Format with Java Format”.

Comparing get and set Methods for oracle.sql.* Format with Java Format

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 of the Oracle SQL types (as
described in "Other getXXX() Methods" on page 4-37), 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 "Additional Type
Extensions" on page 4-111), 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
Data Access and Manipulation: Oracle Types vs. Java Types

database SQL types and are discussed in "Understanding Type Maps" on page 4-66.) 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 4–3, "Summary of `getObject()` and `getOracleObject()` Return Types" on page 4-36.

**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 `Datum` object. The prototype for the method is:

```java
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 4–3, "Summary of `getObject()` and `getOracleObject()` Return Types" on page 4-36.

**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 gets 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.

```java
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 string_table");
while (rset.next ()){
    CHAR char_datum = (CHAR) ((OracleResultSet)rset).getOracleObject (1);
    ...
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 since getOracleObject() returns a Datum object, the results are cast to a DATE object.

```java
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 4–3 summarizes the information in the preceding sections, "Standard getObject() Method" on page 4-34 and "Oracle getOracleObject() Method" on page 4-35.

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`
- `getOracleObject()` always returns data into an `oracle.sql.Datum`

You must cast the returned object to use any special functionality (see "Casting Your `get` Method Return Values" on page 4-39).

<table>
<thead>
<tr>
<th>Oracle SQL Type</th>
<th><code>getObject()</code> Underlying Return Type</th>
<th><code>getOracleObject()</code> Underlying Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>String</td>
<td><code>oracle.sql.CHAR</code></td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>String</td>
<td><code>oracle.sql.CHAR</code></td>
</tr>
</tbody>
</table>
For information on type compatibility between all SQL and Java types, see Table 8–1, "Valid SQL Datatype-Java Class Mappings" on page 8-2.

**Other getXXX() Methods**

Standard JDBC provides a `getXXX()` for each standard Java type, such as `getByte()`, `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 of the `oracle.sql.*` types. Each `getXXX()` method returns an `oracle.sql.XXX`. For example, `getROWID()` returns an `oracle.sql.ROWID`.

---

**Table 4–3 Summary of getObject() and getOracleObject() Return Types (Cont.)**

<table>
<thead>
<tr>
<th>Oracle SQL Type</th>
<th><code>getObject()</code> Underlying Return Type</th>
<th><code>getOracleObject()</code> Underlying Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LONG</td>
<td>String</td>
<td><code>oracle.sql.CHAR</code></td>
</tr>
<tr>
<td>NUMBER</td>
<td><code>java.math.BigDecimal</code></td>
<td><code>oracle.sql.NUMBER</code></td>
</tr>
<tr>
<td>RAW</td>
<td><code>byte[]</code></td>
<td><code>oracle.sql.RAW</code></td>
</tr>
<tr>
<td>LONGRAW</td>
<td><code>byte[]</code></td>
<td><code>oracle.sql.RAW</code></td>
</tr>
<tr>
<td>DATE</td>
<td><code>java.sql.Timestamp</code></td>
<td><code>oracle.sql.DATE</code></td>
</tr>
<tr>
<td>ROWID</td>
<td><code>oracle.sql.ROWID</code></td>
<td><code>oracle.sql.ROWID</code></td>
</tr>
<tr>
<td>REF CURSOR</td>
<td><code>java.sql.ResultSet</code></td>
<td>(not supported)</td>
</tr>
<tr>
<td>BLOB</td>
<td><code>oracle.sql.BLOB</code></td>
<td><code>oracle.sql.BLOB</code></td>
</tr>
<tr>
<td>CLOB</td>
<td><code>oracle.sql.CLOB</code></td>
<td><code>oracle.sql.CLOB</code></td>
</tr>
<tr>
<td>BFILE</td>
<td><code>oracle.sql.BFILE</code></td>
<td><code>oracle.sql.BFILE</code></td>
</tr>
<tr>
<td>Oracle object</td>
<td>class specified in type map OR</td>
<td><code>oracle.sql.STRUCT</code></td>
</tr>
<tr>
<td></td>
<td><code>oracle.sql.STRUCT</code> (if no type map entry)</td>
<td></td>
</tr>
<tr>
<td>Oracle object reference</td>
<td><code>oracle.sql.REF</code></td>
<td><code>oracle.sql.REF</code></td>
</tr>
<tr>
<td>collection (varray or</td>
<td><code>oracle.sql.ARRAY</code></td>
<td><code>oracle.sql.ARRAY</code></td>
</tr>
<tr>
<td>nested table)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some of these extensions are taken from the JDBC 2.0 specification. They return objects of type oracle.jdbc2.* instead of oracle.sql.*. For example, compare the prototypes:

```java
oracle.jdbc2.Blob getBlob(int parameter_index)
```

which returns an oracle.jdbc2 type for BLOBs, in contrast to:

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

which returns an oracle.sql type for BLOBs.

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.

Table 4–4 summarizes the underlying return types and the signature types for each `getXXX()` method. You must cast to an OracleResultSet or OracleCallableStatement to use methods that are Oracle-specific.

<table>
<thead>
<tr>
<th>Method</th>
<th>Underlying Return Type</th>
<th>Signature Type</th>
<th>Oracle Specific?</th>
</tr>
</thead>
<tbody>
<tr>
<td>getArray()</td>
<td>oracle.sql.ARRAY</td>
<td>oracle.jdbc2.Array</td>
<td>Yes</td>
</tr>
<tr>
<td>getARRAY()</td>
<td>oracle.sql.ARRAY</td>
<td>oracle.sql.ARRAY</td>
<td>Yes</td>
</tr>
<tr>
<td>getBfile()</td>
<td>oracle.sql.BFILE</td>
<td>oracle.sql.BFILE</td>
<td>Yes</td>
</tr>
<tr>
<td>getBFILE()</td>
<td>oracle.sql.BFILE</td>
<td>oracle.sql.BFILE</td>
<td>Yes</td>
</tr>
<tr>
<td>getBigDecimal()</td>
<td>BigDecimal</td>
<td>BigDecimal</td>
<td>No</td>
</tr>
<tr>
<td>getBlob()</td>
<td>oracle.sql.BLOB</td>
<td>oracle.jdbc2.Blob</td>
<td>Yes</td>
</tr>
<tr>
<td>getBLOB</td>
<td>oracle.sql.BLOB</td>
<td>oracle.sql.BLOB</td>
<td>Yes</td>
</tr>
<tr>
<td>getBoolean()</td>
<td>boolean</td>
<td>boolean</td>
<td>No</td>
</tr>
<tr>
<td>getByte()</td>
<td>byte</td>
<td>byte</td>
<td>No</td>
</tr>
<tr>
<td>getBytes()</td>
<td>byte[]</td>
<td>byte[]</td>
<td>No</td>
</tr>
<tr>
<td>getCHAR()</td>
<td>oracle.sql.CHAR</td>
<td>oracle.sql.CHAR</td>
<td>Yes</td>
</tr>
<tr>
<td>getClob()</td>
<td>oracle.sql.CLOB</td>
<td>oracle.jdbc2.Clob</td>
<td>Yes</td>
</tr>
<tr>
<td>getCLOB()</td>
<td>oracle.sql.CLOB</td>
<td>oracle.sql.CLOB</td>
<td>Yes</td>
</tr>
<tr>
<td>getDate()</td>
<td>java.sql.Date</td>
<td>java.sql.Date</td>
<td>No</td>
</tr>
</tbody>
</table>
Casting Your get Method Return Values

As described in "Standard getObject() Method" on page 4-34, Oracle’s implementation of `getObject()` always returns a `java.lang.Object` and `getOracleObject()` always returns an `oracle.sql.Datum`. 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` as opposed to a `java.lang.Object`.

**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
Data Access and Manipulation: Oracle Types vs. Java Types

To manipulate the CHAR data without losing precision, cast your result set to an OracleResultSet and use getOracleObject() to return the CHAR data. (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.) By casting the result set, you can use getOracleObject() and return data in oracle.sql.* format.

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 later use any 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, return into a generic oracle.sql.Datum return variable and cast this object later whenever 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.

You can use the setObject() method to bind standard Java types to a prepared statement or callable statement; it takes a java.lang.Object as input. You can use the setOracleObject() method to bind oracle.sql.* types; it takes an oracle.sql.Datum (or any subclass) as input. The setObject() method supports some oracle.sql.* types—see note below. For other oracle.sql.* types, you must use setOracleObject().
To use `setOracleObject()`, you must cast your prepared statement or callable statement to an `OraclePreparedStatement` or `OracleCallableStatement` object.

---

**Note:** The `setObject()` method 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`.

---

**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. Since 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, since you want to manipulate the data in column 2 as strings, cast the data to a Java `String` type (since `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.

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

For some prepared statement `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`.

```java
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.

In addition, for compatibility with the JDBC 2.0 standard, `OraclePreparedStatement` and `OracleCallableStatement` classes provide `setXXX()` methods that take `oracle.jdbc2` input parameters for `BLOBs`, `CLOBs`, `BFILEs`, `STRUCTs`, `REFs`, and `ARRAYs`.
object references, and arrays. For example, there is a `setBlob()` method that takes an `oracle.jdbc2.Blob` input parameter, and a `setBLOB()` method that takes an `oracle.sql.BLOB` input parameter.

Similarly, there are two forms of the `setNull()` method:

- `void setNull(int parameterIndex, int sqlType)`
  behaves in a similar way to the standard Java `java.sql.PreparedStatement.setNull()`. This method takes a parameter index and a SQL type code defined by `java.sql.Types`. You use this method to set an object (except for REFs, ARRAYS, or STRUCTs) to NULL.

- `void setNull(int parameterIndex, int sqlType, String sql_type_name)`
  takes a SQL type name in addition to a parameter index and a SQL type code. You use this method only when the SQL type code is REF, ARRAY, or STRUCT.

Similarly, the `OracleCallableStatement.registerOutParameter()` method also has an overloaded method that you use when working with REFs, ARRAYS, or STRUCTs.

- `void registerOutParameter(int parameterIndex, int sqlType, String sql_type_name)`

There is no particular performance advantage in using the specific `setXXX()` methods for binding Oracle-specific types over the methods for binding standard Java types.

Table 4–5 summarizes the input types for all of the `setXXX()` methods. To use methods that are Oracle-specific, you must cast your statement to an `OraclePreparedStatement` or `OracleCallableStatement`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Input Parameter Type</th>
<th>Oracle Specific?</th>
</tr>
</thead>
<tbody>
<tr>
<td>setArray()</td>
<td>oracle.jdbc2.Array</td>
<td>Yes</td>
</tr>
<tr>
<td>setARRAY()</td>
<td>oracle.sql.ARRAY</td>
<td>Yes</td>
</tr>
<tr>
<td>setBfile()</td>
<td>oracle.sql.BFILE</td>
<td>Yes</td>
</tr>
<tr>
<td>setBFILE()</td>
<td>oracle.sql.BFILE</td>
<td>Yes</td>
</tr>
<tr>
<td>setBigDecimal()</td>
<td>BigDecimal</td>
<td>No</td>
</tr>
<tr>
<td>setBlob()</td>
<td>oracle.jdbc2.Blob</td>
<td>Yes</td>
</tr>
<tr>
<td>setBLOB()</td>
<td>oracle.sql.BLOB</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Table 4–5  Summary of setXXX() Input Parameter Types (Cont.)

<table>
<thead>
<tr>
<th>Method</th>
<th>Input Parameter Type</th>
<th>Oracle Specific?</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBoolean()</td>
<td>boolean</td>
<td>No</td>
</tr>
<tr>
<td>setByte()</td>
<td>byte</td>
<td>No</td>
</tr>
<tr>
<td>setBytes()</td>
<td>byte[]</td>
<td>No</td>
</tr>
<tr>
<td>setCHAR()</td>
<td>oracle.sql.CHAR</td>
<td>Yes</td>
</tr>
<tr>
<td>setClob()</td>
<td>oracle.jdbc2.Clob</td>
<td>Yes</td>
</tr>
<tr>
<td>setCLOB()</td>
<td>oracle.sql.CLOB</td>
<td>Yes</td>
</tr>
<tr>
<td>setDate()</td>
<td>java.sql.Date</td>
<td>No</td>
</tr>
<tr>
<td>setDATE()</td>
<td>oracle.sql.DATE</td>
<td>Yes</td>
</tr>
<tr>
<td>setDouble()</td>
<td>double</td>
<td>No</td>
</tr>
<tr>
<td>setFloat()</td>
<td>float</td>
<td>No</td>
</tr>
<tr>
<td>setInt()</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>setLong()</td>
<td>long</td>
<td>No</td>
</tr>
<tr>
<td>setNUMBER()</td>
<td>oracle.sql.NUMBER</td>
<td>Yes</td>
</tr>
<tr>
<td>setRAW()</td>
<td>oracle.sql.RAW</td>
<td>Yes</td>
</tr>
<tr>
<td>setRef()</td>
<td>oracle.jdbc2.Ref</td>
<td>Yes</td>
</tr>
<tr>
<td>setREF()</td>
<td>oracle.sql.REF</td>
<td>Yes</td>
</tr>
<tr>
<td>setROWID()</td>
<td>oracle.sql.ROWID</td>
<td>Yes</td>
</tr>
<tr>
<td>setShort()</td>
<td>short</td>
<td>No</td>
</tr>
<tr>
<td>setString()</td>
<td>String</td>
<td>No</td>
</tr>
<tr>
<td>setSTRUCT()</td>
<td>oracle.sql.STRUCT</td>
<td>Yes</td>
</tr>
<tr>
<td>setTime()</td>
<td>java.sql.Time</td>
<td>No</td>
</tr>
<tr>
<td>setTimestamp()</td>
<td>java.sql.Timestamp</td>
<td>No</td>
</tr>
</tbody>
</table>

For information on type compatibility between all SQL and Java types, see Table 8–1, "Valid SQL Datatype-Java Class Mappings" on page 8-2.
Using Result Set Meta Data Extensions

Although the oracle.jdbc.driver.OracleResultSetMetaData class does not implement the full JDBC 2.0 API for retrieving result set meta data, it does provide many methods to retrieve information about an Oracle result set.

The getColumnTypeName() method takes a column number and returns the SQL type name for columns of type REF, STRUCT, or ARRAY. In contrast, the getColumnType() method takes a column number and returns the SQL type. If the column stores an Oracle object or collection, then it returns an OracleTypes.STRUCT or an OracleTypes.ARRAY. For a list of the key methods provided by OracleResultSetMetadata, see "Class oracle.jdbc.driver.OracleResultSetMetadata" on page 4-28.

The following example uses several of the methods in the OracleResultSetMetadata class to retrieve the number of columns from the EMP table, and each column’s numerical type and SQL type name.

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

while (rset.next())
{
    OracleResultSetMetaData orsmd = ((OracleResultSet)rset).getMetaData();
    int numColumns = orsmd.getColumnCount();
    System.out.println("Num of columns = " + numColumns);
    for (int i=0; i<numColumns; i++)
    {
        System.out.print ("Column Name=" + orsmd.getColumnName (i+1));
        System.out.print (" Type=" + orsmd.getColumnType (i + 1) );
        System.out.println (" Type Name=" + orsmd.getColumnTypeName (i + 1));
    }
}
```

The program returns the following output:

```
Num of columns = 5
Column Name=TABLE_CAT Type=12 Type Name=VARCHAR2
Column Name=TABLE_SCHEM Type=12 Type Name=VARCHAR2
Column Name=TABLE_NAME Type=12 Type Name=VARCHAR2
Column Name=TABLE_TYPE Type=12 Type Name=VARCHAR2
Column Name=TABLE_REMARKS Type=12 Type Name=VARCHAR2
```
Working with LOBs

This section has these subsections:

- Getting BLOB and CLOB Locators
- Passing BLOB and CLOB Locators
- Reading and Writing BLOB and CLOB Data
- Creating and Populating a BLOB or CLOB Column
- Accessing and Manipulating BLOB and CLOB Data
- Getting BFILE Locators
- Passing BFILE Locators
- Reading BFILE Data
- Creating and Populating a BFILE Column
- Accessing and Manipulating BFILE Data

LOBs can be either internal or external. Internal LOBs, as their name suggests, are stored inside database tablespaces in a way that optimizes space and provides efficient access. The JDBC drivers provide support for two types of internal LOBs: BLOBs (unstructured binary data) and CLOBs (single-byte 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.

External LOBs (BFILES) are large binary data objects stored in operating system files outside of database tablespaces. These files use reference semantics. They may also be located on tertiary storage devices such as hard disks, CD-ROMs, PhotoCDs and DVDs. Like 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.

This section describes how you use JDBC and the oracle.sql.* classes to work with LOBs. To work with LOB data, you must first obtain its locator from the table. Then, you can read data from or write data to the LOB and perform various types of data manipulation. This section also describes 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
Working with LOBs

- `oracle.sql.BFILE`

The `oracle.sql.BLOB` and `CLOB` classes implement the `oracle.jdbc2.Blob` and `Clob` interfaces, respectively. In contrast, `BFILEs` have no `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 4-48 and "Reading BFILE Data" on page 4-57.

### Getting BLOB and CLOB Locators

Given a standard JDBC result set or callable statement object that includes BLOB or CLOB locators, you can access the locators by using the standard `ResultSet.getObject()` method. This method returns an `oracle.sql.BLOB` object or `oracle.sql.CLOB` object, as applicable (but note that it returns the BLOB or CLOB into a variable of type `oracle.jdbc2.Blob` or `oracle.jdbc2.Clob`).

You can also access the locators by casting your result set to `OracleResultSet` or your callable statement to `OracleCallableStatement` and using the `getOracleObject()`, `getBLOB()`, or `getCLOB()` method, as appropriate.

In the `OracleResultSet` and `OracleCallableStatement` classes, `getBlob()` returns `oracle.jdbc2.Blob`, and `getBLOB()` returns `oracle.sql.BLOB`. Similarly, `getCLOB()` returns `oracle.jdbc2.CLOB` and `getClob()` returns `oracle.sql.Clob`.

---

**Notes:**

- 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 4-39.
- Refer the Javadoc for more information about specific features of the `oracle.sql.BLOB` and `oracle.sql.CLOB` classes.

---

**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:

```java
// 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.
    oracle.jdbc2.Blob blob = (oracle.jdbc2.Blob)rs.getObject(1);
    oracle.jdbc2.Clob clob = (oracle.jdbc2.Clob)rs.getObject(2);
    [...process...]
}
```

The output is cast to `oracle.jdbc2.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:

```java
// 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. In the case of a callable statement, register the output parameter as `OracleTypes.BLOB` or `OracleTypes.CLOB`.

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 follows:

```java
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("(? = call func())");
ocs.registerOutParameter(1, OracleTypes.CLOB);
ocs.executeQuery();
oracle.sql.CLOB clob = ocs.getCLOB(1);
```

**Passing BLOB and CLOB Locators**

To pass a LOB locator to a prepared statement or callable statement (to update a LOB locator in the database, for example), you can use the generic `setObject()` method, or you can cast the statement to `OraclePreparedStatement` or `OracleCallableStatement` and use the `setOracleObject()`, `setBLOB()`, or
setCLOB() method, as appropriate. These methods take the parameter index and a BLOB object or CLOB object as input.

**Example: Passing a BLOB Locator to a Prepared Statement** If you have an OraclePreparedStatement ops where its first parameter is a BLOB named my_blob, then input the BLOB to the prepared statement as follows:

```java
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 ocs where its first parameter is a CLOB, then input the CLOB to the callable statement as follows:

```java
OracleCallableStatement ocs = (OracleCallableStatement) conn.prepareCall("? := call func()")
ocs.setClob(1, my_clob)
ocs.execute();
```

**Reading and Writing BLOB and CLOB Data**

The SQL SELECT statement queries for LOB locators. Once you have the locator, you can read and write the LOB data from JDBC. LOB data is materialized as a Java 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.
Working with LOBs

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.

Notes:

- The implementation of the data access API uses direct native calls in the JDBC OCI and Server drivers, thereby providing better performance. You can use the same API on the LOB classes in all Oracle 8.1.5 JDBC drivers.

- In the case of the JDBC Thin driver only, the implementation of the data access API uses the `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 *Oracle8i Application Developer’s Guide - Large Objects (LOBs)* and the *Oracle8i Application Developer’s Reference - Packages*.

To read and write LOB data, you can use these methods:
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 `close()` method 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/COMMIT` to write the data.
- When writing to or reading from a CLOB, the JDBC drivers handle all character set conversions for you.

**Example: Reading BLOB Data**  Use the `getBinaryStream()` method of the `oracle.sql.BLOB` class to read BLOB data. The `getBinaryStream()` method 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).

```java
// 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 object in an oracle.sql.CLOB, not an oracle.jdbc2.Clob.

```
java.io.Writer writer

// read data into a character array
Working with LOBs

```java
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 an `oracle.sql.CLOB` datatype.

```java
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

You create and populate a BLOB or CLOB column in a table by using SQL statements.

**Note:** To create a BLOB or CLOB column in a table, you must use SQL statements. Using the Java `new`, such as `new BLOB` or `new CLOB` will not work.

You create a BLOB or CLOB column in a table with the SQL `CREATE TABLE` statement. Then, you 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:

```java
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 one or two, 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.
   ```java
   stmt.execute ("insert into my_blob_table values ('row1', empty_blob())");
   ```

2. Get the BLOB locator from the table.
   ```java
   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.
   ```java
   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 getChunkSize() to determine the ideal chunk size to write to the BLOB, then create the buffer byte array.
   ```java
   int chunk = blob.getChunkSize();
   byte[] buffer = new byte[chunk];
   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.

```java
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 following 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 BLOB and CLOB Locators" on page 4-46 describes these techniques in detail.

After you select the locators, you can get 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 getting 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.

```java
// Select the blob - what we are really doing here
// is getting the blob locator into a result set
BLOB blob;
String 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.getlength();

// 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(0, length);
printBytes(bytes, length);

---

**Getting 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 `ResultSet.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 `oracle.jdbc2` class for BFILE.
- If using `getObject()` or `getOracleObject()`, remember to cast the output, as necessary. For more information, see "Casting Your get Method Return Values" on page 4-39.

---

**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 data:

```java
// 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, since `getObject()` returns a `java.lang.Object`, cast the results to `BFILE`. For example:

```java
oracle.sql.BFILE my_bfile = (BFILE) rs.getObject(1);
```

**Example: Getting a BFILE Locator from a Callable Statement**

Assume you have an `OracleCallableStatement` `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:

```java
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 use the generic `setObject()` method or you can 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**

You want to insert a BFILE locator into a table. Assume that you have an `OraclePreparedStatement` `ops` where the first parameter is a string (to designate a row number), its second parameter is a BFILE, and you have a valid `oracle.sql.BFILE` object (`bfile`). Input the BFILE to the prepared statement as follows:

```java
OraclePreparedStatement ops =
    (OraclePreparedStatement) conn.prepareStatement
    ("INSERT INTO my_bfile_table VALUES (?,?)");
ops.setString(1, "one");
ops.setBFILE(2, bfile);
ops.executeUpdate();
```

**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);
   }
   finally
   {
      cstmt.close ();
   }

**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 BFILE Locators" on page 4-55 describes this.

Once you obtain the locator, there are a number of methods that you can perform on the BFILE without opening it. For example, you can use the oracle.sql.BFILE methods fileExists() and isOpen() to determine whether the BFILE exists and if it is open. However, if you want to read and manipulate the data, you must open the BFILE. BFILE data is materialized as a Java stream. Operate on BFILEs from JDBC as follows:

- 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 class.

  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.
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.

```java
// 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

You create a BFILE column in a table with SQL statements 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 location where the BFILE resides is `/home/work`.

```sql
String cmd;
// Create a table containing a BFILE field
cmd = "CREATE TABLE my_bfile_table (x varchar2 (30), b bfile)";
stmt.execute (cmd);
```

Notes:

- BFILEs are read-only. You cannot insert data or otherwise write to a BFILE.
- You cannot use JDBC to create a new BFILE.
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 keyword to specify the directory alias and the name of the BFILE file.

```sql
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.

```sql
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:

```java
rs = stmt.executeQuery("SELECT b FROM my_bfile_table WHERE x='two'");
rs.next()
oracle.sql.BFILE bfile = ((OracleResultSet)rs).getBFILE(2);
```

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`:

```java
OraclePreparedStatement ops = (OraclePreparedStatement)conn.prepareStatement("INSERT INTO my_bfile_table WHERE (x = 'three')
    ...
ops.setBFILE(2, bfile);
```

In this example, the BFILE 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.
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 example gets 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 "BFILE Sample" on page 7-10.

```java
// select the bfile locator
cmd = "SELECT * FROM my_bfile_table WHERE x = 'two'";
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] );
    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();
Working with Oracle Object Types

This section contains these subsections:

- Using Default Java Classes for Oracle Objects
- Creating Custom Java Classes for Oracle Objects
- Using JPublisher with JDBC

Oracle object types provide support for composite data structures in the database. For example, you could define a type `Person` that has attributes such as name (type `CHAR`), address (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 customize how SQL types map to Java classes by creating custom Java type definition classes; Oracle offers considerable flexibility in how this mapping is done. In this book, Java classes created as classes to map to Oracle objects will be referred to as custom Java classes.

JDBC materializes Oracle objects as instances of particular Java classes. Two main steps in using JDBC to access Oracle objects are: creating the Java classes for the Oracle objects and populating these classes. You have the option of:

- letting JDBC materialize the object as a `Struct`. This is described in "Using Default Java Classes for Oracle Objects" on page 4-62.

OR

- explicitly specifying 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 the custom Java classes that you specify. This imposes a set of constraints on the Java classes. To satisfy these constraints, you can define your classes according to either the `SQLData` interface or the `CustomDatum` interface. "Creating Custom Java Classes for Oracle Objects" on page 4-65 describes this.

Using Default Java Classes for Oracle Objects

If you choose not to provide a type map to explicitly specify a Java class for an Oracle object, you can let Oracle JDBC materialize the object as a `Struct`.

You would typically want to use `Struct` objects instead of custom Java objects in situations where you are manipulating data. For example, your Java application might be a tool to manipulate data 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. As described in "Class oracle.sql.STRUCT" on page 4-10, STRUCTs completely preserve data because they maintain the data in SQL format. Using Struct objects is more efficient and more precise in these situations where the information does not need to be in a user-friendly format.

If your code must fully comply with JDBC 2.0, use the functionality in the oracle.jdbc2.Struct interface:

- `getAttributes(map)`: retrieves the values from the values array as `java.lang.Object` objects; uses entries in the type map (if they have been defined) to determine the Java classes to use in materializing the data.
- `getAttributes()`: retrieves the values of the values array as `java.lang.Object` objects
- `getSQLTypeName()`: returns a Java `String` that represents the fully qualified type name (`schema.sql_type_name`) of the Oracle object that this Struct represents

If it is not necessary to comply with JDBC 2.0 and you want to take advantage of the extended functionality offered by Oracle-defined methods, then cast the output to oracle.sql.STRUCT.

The oracle.sql.STRUCT class implements the oracle.jdbc2.Struct interface and provides extended functionality beyond the JDBC 2.0 standard. Compare the list of methods above with the methods provided for oracle.sql.STRUCT in "Class oracle.sql.STRUCT" on page 4-10.

### Using STRUCT Objects

You can use standard JDBC functionality such as `getObject()` to retrieve Oracle objects from the database as an instance of oracle.jdbc2.Struct. Because `getObject()` returns a `java.lang.Object`, you must cast the output of the method to a Struct. For example:

```java
oracle.jdbc2.Struct myStruct = (oracle.jdbc2.Struct)rs.getObject(1);
```

As described in the preceding section, the oracle.jdbc2.Struct class is implemented by oracle.sql.STRUCT. If you want to use the extended functionality offered by Oracle, you can then cast the Struct object to a STRUCT. For example, to use the `getOracleAttributes()` method to return the attributes of the Struct, cast `myStruct` to oracle.sql.STRUCT:

```java
oracle.sql.STRUCT STRUCTattribute=s
```
The `getOracleAttributes()` method returns the attributes of `myStruct` in `oracle.sql.*` format.

You can also retrieve the object directly into an `oracle.sql.STRUCT`. For 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 result is cast to an `oracle.sql.STRUCT`. This example assumes that the `Statement` object `stmt` has already been created.

```java
String cmd;
stmt.execute(cmd);
```

Similarly, to get data from the database, the `OracleCallableStatement` and `OracleResultSet` classes have a `getSTRUCT()` method that returns an Oracle object as an `oracle.sql.STRUCT`. For example:

```java
stmt.executeQuery("SELECT * FROM test_Struct");
oracle.sql.STRUCT myStruct = ((OracleResultSet)rs).getSTRUCT();
```
Creating Custom Java Classes for Oracle Objects

If you want to define custom Java classes for your Oracle objects, then you must define a type map that specifies the custom Java classes that the drivers will generate for the corresponding Oracle objects.

You must also provide a way to create and populate the custom Java class from the Oracle object and its attribute data. The driver must be able to read from a Java custom class and populate it. In addition, the custom Java class can provide get and set 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:

- SQLData interface provided by JDBC
- CustomDatum interface provided by Oracle

The custom Java class you create must implement one of these interfaces.

For 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 Java class that you call JEmployee. You can use either the SQLData or CustomDatum interface to be implemented by the JEmployee class.

The most convenient way to create the custom Java class is to employ the JPublisher utility to create it for you. However, JPublisher supports only the CustomDatum implementation. You can also create the custom Java class yourself, and in fact must do so if you want to implement the SQLData interface.

The following section describes the relative advantages of using CustomDatum and SQLData.

Relative Advantages of CustomDatum vs. SQLData In deciding which of these two interface implementations to use, consider the following:

Advantages of CustomDatum:

- It has awareness of Oracle extensions.
- You can construct a CustomDatum 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 CustomDatum object using the toDatum() method.
■ It does not require a type map.
■ It provides better performance: CustomDatum works directly with Datum types, which is the internal format used by the driver to hold Oracle objects.
■ JPublisher supports it. Custom Java classes created by JPublisher use the CustomDatum implementation. As of the 8.1.5 release, SQLData is not supported by JPublisher.
■ Oracle SQLJ supports it. As of the 8.1.5 release, SQLData is not supported by Oracle’s implementation of SQLJ.

Advantages of SQLData:
■ It is a JDBC standard, making your code more portable.

The SQLData interface only lets you populate a Java object from a SQL object—the CustomDatum interface is far more powerful. In addition to enabling you to populate Java objects, CustomDatum enables you to materialize objects from SQL types that are not necessarily objects. Therefore, you can create a CustomDatum object from any datatype found in an Oracle database. This is particularly useful in the case of RAW data that can be a serialized object.

Understanding Type Maps
If you use the SQLData interface to create Java custom classes, then you must create a type map that specifies the Java custom class that corresponds to the Oracle object in the database. For a description of how to create these custom Java classes with SQLData, see "Creating Custom Java Classes for Oracle Objects" on page 4-65.

If you do not include an object and its mapping in the type map, then the object will map to the oracle.sql.STRUCT class by default. See "Class oracle.sql.STRUCT" on page 4-10 for more information about this class.

The type map relates a Java class to the SQL type name of an Oracle object. This is a one-to-one mapping that is stored in a hash table as a key-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 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 handled 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 Class

The Java application programmer is responsible for providing a type map class that implements `java.util.Dictionary`. For example, `java.util.Hashtable` implements `Dictionary`.

The type map class must implement a `put()` method used to enter each mapping entry that relates a Java class to an Oracle object type. The `put()` method must be implemented to accept a `keyword-value` pair, where the key is an Oracle SQL type name and the value is the Java class object.

Creating a Type Map Object and Defining Mappings

Each connection object has an attribute for an associated type map object. Cast your connection to an `OracleConnection` object to use type map functionality.

You can create a type map by either of the methods described in the following sections:

- Adding Entries to an Existing Type Map
- Creating a New Type Map

Adding Entries to an Existing Type Map

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 `Map` object. The `getTypeMap()` method returns a `java.util.Dictionary` object. For example:

   ```java
   java.util.Dictionary myMap = oraconn.getTypeMap();
   ```

   In this example, the `getMapType()` method on the `OracleConnection` object `oraconn` returns the `myMap` `Dictionary` object.

   **Note:** If the type map in the `OracleConnection` object has not been initialized, then the first call to `getTypeMap()` returns `null`.

2. Use the `Dictionary` object's `put()` method to add entries to the map. The `put()` method takes two arguments: a SQL type name string and the name of the Java class object to which you want to map it.

   ```java
   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. You can rewrite the put() method as:

\[\text{myMap.put(sqlTypeName, class.forName(className))}\]

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:

\[\text{myMap.put("CORPORATE.PERSON", class.forName("Person"))}\]

The map has an entry that maps the PERSON SQL datatype in the CORPORATE database to the Person Java class.

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:

\[\text{oraconn.setFactoryMap(myMap)}\]

In this example, setTypeMap() overwrites the oraconn connection’s original map with myMap.

Creating a New Type Map Follow these general steps to create a new type map.

1. Create an empty map object. An empty map object can be anything that implements the java.util.Dictionary class. For example, the java.util.Hashtable class implements the Dictionary class.

2. Use the Map object’s put() method to add entries to the map. For more information on the put() method, see Step 2 in the preceding section. 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:

\[\text{newMap.put("CORPORATE.EMPLOYEE", class.forName("Employee"))}\]

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:

\[\text{oraconn.setFactoryMap(newMap)}\]
In this example, `setTypeMap()` overwrites the `oraconn` connection’s original map with `newMap`.

---

**Notes:**

- You can explicitly provide type map objects in some `getXXX()` and `setXXX()` methods to override the custom or default mapping of your connection.
- If the type map does not specify a Java class mapping for an Oracle object type, then it defaults to data from the object type materialized in Java in an instance of the `oracle.sql.STRUCT` class. For more information about this class, see “Class `oracle.sql.STRUCT`” on page 4-10.
- Do not use the type map for inserting custom objects into the database.

---

**STRUCTS and the Type Map** If you do not specify a particular SQL object type in the type map, then the driver will materialize it as an instance of the `oracle.jdbc2.Struct` class. If the SQL 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`. 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**

To make an Oracle object and its attribute data available to Java applications, you can create a custom Java class for the object that implements the `SQLData` interface. Note that if you use this interface, you must supply a type map that specifies the Oracle objects in the database and the name of the corresponding custom Java 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 `oracle.jdbc2` package.

If you create a custom Java class that implements `SQLData`, you must provide a `readSQL()` method and a `writeSQL()` method as defined 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 Java class. Typically, the driver would use this method as part of an `OracleResultSet.getObject()` call.

Similarly, the JDBC driver calls your `writeSQL()` method to write a sequence of data values from an instance of your custom Java class to a stream that can be written to the database. Typically, the driver would use this method as part of an `OraclePreparedStatement.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 `readSQL()`. `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 `writeSQL()`. `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 your custom Java class that implements `SQLData`, you must also implement the `readSQL()` and `writeSQL()` methods.

You must implement `readSQL()` as follows:

```java
public void readSQL(SQLInput stream, String sql_type_name) throws SQLException
```

- `readSQL()` must take 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.

`readSQL()` assigns the data that the `readXXX()` methods read and convert to the appropriate fields or elements of your custom Java class.

You must implement `writeSQL()` as follows:

```java
public void writeSQL(SQLOutput stream) throws SQLException
```

- `writeSQL()` must take 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 your custom Java class. 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.

- `writeSQL()` must then write the data converted by the `writeXXX()` methods to the `SQLOutput` stream so it can be written to the database once you execute the prepared statement.

---

**Note:** Refer to the Javadoc for more information about the `SQLData`, `SQLInput`, and `SQLOutput` interfaces.

"Creating Customized Java Classes for Oracle Objects" on page 7-20 contains an example implementation of the `SQLData` interface for a given SQL definition of an Oracle object.
Reading and Writing Data with a SQLData Class

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 Data from an Oracle Object Using a SQLData Interface

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 Java class.

These steps assume you have already defined the Oracle object type, created the corresponding custom Java 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.

   ```java
   ResultSet rs = stmt.executeQuery("SELECT Emp_col FROM PERSONNEL");
   rs.next();
   
   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 Java 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.

   ```java
   Employee emp = (Employee)rs.getObject(1);
   
   Note that if the type map did not have an entry for the object, getObject() would return an oracle.sql.STRUCT object. In this case you must cast the output to an oracle.sql.STRUCT.

   ```java
   Struct empstruct = (oracle.sql.STRUCT)rs.getObject(1);
   ...
   
   The getObject() call triggers readSQL() and readXXX() calls as described above.
3. If you have get methods in your custom Java class, then use them to read data from your object attributes. For example, if EMPLOYEE has an EmpName (employee name) 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 (int). Then invoke them in your Java application as follows:

```java
String empname = emp.getName();
int empnumber = emp.getEmpNum();
```

**Note:** To avoid the need for the type map, use the getSTRUCT() method. This method always returns a STRUCT object even if there is a mapping entry in the type map.

---

Passing SQLData Objects to a Callable Statement as an 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.

   ```java
   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. The SQL type of the Employee object is OracleTypes.STRUCT. Then, execute the statement.

   ```java
   ocs.setInt(2, empnumber);
   ocs.registerOutParameter(1, OracleTypes.STRUCT, "EMP_OBJECT");
   ocs.execute();
   ```

3. Use the getObject() method to retrieve the employee object. Because the object is returned as a STRUCT, cast the output of getObject() to an Employee object.

**Note:** Alternatively, fetch data by using a callable statement object, which also has a getObject() method.

---

Note: To avoid the need for the type map, use the getSTRUCT() method. This method always returns a STRUCT object even if there is a mapping entry in the type map.
Employee emp = (Employee) 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.

   ```java
   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.

   ```java
   ocs.setObject(1, emp);
   ocs.execute();
   ```

**Writing Data to an Oracle Object Using a SQLData Interface** 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 Java 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 Java class, then use them to write data from Java variables in your application to attributes of your Java datatype object.

   ```java
   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 SQLData Interface" on page 4-72.

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.

   ```java
   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.

```java
pstmt.setObject(1, emp);
```

4. Execute the statement, which updates the database.

```java
pstmt.executeUpdate();
```

**Note:** You can use your Java datatype objects as either IN or OUT bind variables.

---

**Understanding the CustomDatum Interface**

To make an Oracle object and its attribute data available to Java applications, you can create a custom Java class for the object that implements the `oracle.sql.CustomDatum` and `oracle.sql.CustomDatumFactory` interfaces. The `CustomDatum` and `CustomDatumFactory` 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 `CustomDatum` and `CustomDatumFactory` interfaces.

The `CustomDatum` interface has these additional advantages:

- Recognizes Oracle extensions to the JDBC; `CustomDatum` uses `oracle.sql.Datum` types directly
- Does not require a type map to specify the names of the Java custom classes you want to create
- Provides better performance: `CustomDatum` works directly with `Datum` types, the internal format the driver uses to hold Oracle objects

The `CustomDatum` and `CustomDatumFactory` interfaces do the following:

- The `toDatum()` method of the `CustomDatum` class transforms the data into an `oracle.sql.*` representation.
- `CustomDatumFactory` specifies a `create()` method equivalent to a constructor for your custom Java class. It creates and returns a `CustomDatum` instance. The JDBC driver uses the `create()` method to return an instance of
the custom Java class to your Java application or applet. It takes as input an
oracle.sql.Datum object and an integer indicating the corresponding SQL
type code as specified in the OracleTypes class.

CustomDatum and CustomDatumFactory have the following definitions:

```java
public interface CustomDatum {
    Datum toDatum (OracleConnection conn) throws SQLException;
}
```

```java
public interface CustomDatumFactory {
    CustomDatum create (Datum d, int sql_Type_Code) throws SQLException;
}
```

where conn represents the Connection object, d represents an object of type
oracle.sql.Datum and sql_Type_Code represents the SQL type code of the
Datum object.

**Note:** It is up to the developer to decide how to handle a situation
where the SQL type code contradicts the type of the Datum object.

The JDBC drivers provide the following methods to retrieve and insert object data
as instances of CustomDatum.

To retrieve object data:

- **Use the Oracle extension OracleResultSet.getCustomDatum () method:**

  ```java
  OracleResultSet.getCustomDatum (int col_index, CustomDatumFactory factory)
  ```

  This method takes as input the column index of the data in your result set, and
  a CustomDatumFactory instance. For example, you can implement a
  getFactory () method of your custom Java class to produce the
  CustomDatumFactory instance to input to getCustomDatum (). The type
  map is not required when using Java classes that implement CustomDatum.

  OR

- **Use the standard ResultSet.getObject (index, map) method to retrieve data as instances of CustomDatum.** 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 extension `OraclePreparedStatement.setCustomDatum()` method:

  ```java
  OraclePreparedStatement.setCustomDatum(int bind_index, CustomDatum 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 JDBC `PreparedStatement.setObject()` method. You can also use this method, in its various forms, to insert `CustomDatum` instances without requiring a type map.

The following sections describe the `getCustomDatum()` and `setCustomDatum()` methods.

To continue the example of an Oracle object `EMPLOYEE`, you might have something like the following in your Java application:

```java
CustomDatum datum = ors.getCustomDatum(1, Employee.getFactory());
```

In this example, `ors` is an Oracle result set, `getCustomDatum()` is a method in the `OracleResultSet` class used to retrieve a `CustomDatum` object, and the `EMPLOYEE` is in column 1 of the result set. The `Employee.getFactory()` call will return a `CustomDatumFactory` to the JDBC driver. The JDBC driver will call `create()` from this object, returning to your Java application an instance of the `Employee` class that is populated with data from the result set.
CustomDatum versus SQLData: Comparison for Serializable Objects

The CustomDatum interface provides far more flexibility than the SQLData interface. The SQLData interface is designed to only let you customize the mapping of SQL object types (that is, Oracle8 object types) to Java types of your choice. Implementing the SQLData interface lets the JDBC driver populate the fields of the customized Java class from the original SQL object data and vice-versa, after performing the appropriate conversions between Java and SQL types.

The CustomDatum interface goes beyond simply supporting the customization of SQL 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.

For example, use CustomDatum to store instances of Java objects that do not correspond to a particular SQL Oracle8 object type in the database in columns of SQL type RAW. The create() method in CustomDatumFactory would have to implement a conversion from an object of type oracle.sql_RAW to the desired Java object. The toDatum() method in CustomDatum would have to implement a conversion from the Java object to an oracle.sql_RAW. 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 CustomDatumFactory'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 CustomDatum.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 CustomDatum 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 CustomDatum. For more information on why classes that implement CustomDatum do not need a type map, see "Understanding the CustomDatum Interface" on page 4-75.

Reading and Writing Data with a CustomDatum Interface

This section describes how to read data from an Oracle object or write data to an Oracle object if your corresponding Java class implements CustomDatum.

Reading Data from an Oracle Object Using the CustomDatum Interface

This section summarizes the steps in reading data from an Oracle object into your Java application. These steps apply whether you implement CustomDatum manually or use JPublisher to produce your custom Java classes.

These steps assume you have already defined the Oracle object type, created the corresponding custom Java 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";
   ors.next();

   where PERSONNEL is a one-column table. The column name is Emp_col of type Employee_object.

2. Use the getCustomDatum() method of your Oracle result set to populate an instance of your custom Java class with data from one row of the result set. The getCustomDatum() method returns an oracle.sql.CustomDatum object, which you can cast to your specific custom Java class.

   Employee emp = (Employee)ors.getCustomDatum(1, Employee.getFactory());

   OR

   CustomDatum datum = ors.getCustomDatum(1, Employee.getFactory());
This example assumes that Employee is the name of your custom Java class and ors is the name of your OracleResultSet object.

If you do not want to use getCustomDatum(), the JDBC drivers let you use the standard JDBC ResultSet.getObject() method to retrieve CustomDatum 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 CustomDatum. The corresponding Factory class is EmployeeFactory, which will implement CustomDatumFactory.

Use this statement to declare the EmployeeFactory entry for your type map:

```java
map.put("EMPLOYEE", Class.forName("EmployeeFactory"));
```

Then use the form of getObject() where you specify the map object:

```java
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():

```java
Employee emp = (Employee) rs.getObject (1);
```

3. If you have get methods in your custom Java class, use them to read data from your object attributes into Java variables in your application. For example, if EMPLOYEE has Name of type CHAR and EmpNum (employee number) of type NUMBER, provide a getName() method that returns a Java string and a getEmpNum() method that returns an integer. Then invoke them in your Java application as follows:

```java
String empname = emp.getName();
int empnumber = emp.getEmpNum();
```

---

**Note:** Alternatively, you can fetch data into a callable statement object. The OracleCallableStatement class also has a getCustomDatum() method.

**Writing Data to an Oracle Object Using the CustomDatum Interface** This section summarizes the steps in writing data to an Oracle object from your Java application.
when you use JPublisher to produce your custom Java class or otherwise choose the CustomDatum implementation.

These steps assume you have already defined the Oracle object type, created the corresponding custom Java class or had JPublisher create it for you.

---

**Note:** The type map is not used when you are performing database INSERTs and UPDATES.

---

1. If you have set methods in your custom Java class, then use them to write data from Java variables in your application to attributes of your Java datatype object.

   ```
   emp.setName(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 the CustomDatum Interface" on page 4-79.

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 `setCustomDatum()` method of the Oracle prepared statement to bind your Java datatype object to the prepared statement.

   ```
   opstmt.setCustomDatum(1, emp);
   ```

   The `setCustomDatum()` method calls the `toDatum()` method of your custom Java class 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);
   ```
Using JPublisher with JDBC

JPublisher is an Oracle utility for creating Java classes that map to Oracle objects. It generates a full class 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 attributes of the class.

If you want additional functionality you can create a subclass and add features as desired. JPublisher has features that will create references to the code you write if you need to regenerate the original class. The alternative, editing the generated class by adding methods to it, is not recommended if you anticipate running JPublisher at some future time to regenerate the class. If you run JPublisher to regenerate a class that you have modified in this way, your changes (that is, the methods you have added) will be overwritten. Even if you direct JPublisher output to a separate file, you will still need to merge your changes into the file.

You do not have to use JPublisher to create your custom Java classes, but it is usually very convenient. For more information on JPublisher, see the Oracle8i JPublisher User's Guide.

JPublisher Mapping Options

If you use JPublisher to implement your custom Java class, then you can choose among three mappings for attributes:

- Oracle mapping
- JDBC mapping
- Object JDBC mapping

JPublisher has a command-line option that enables you to choose among these three mapping options. For more information on the mapping options, see the Oracle8i JPublisher User’s Guide.

Note: You can use your Java datatype objects as either IN or OUT bind variables.
Working with Oracle Object References

This section has these subsections:

- Retrieving an Object Reference
- Passing an Object Reference to a Callable Statement
- Accessing and Updating Object Values through an Object Reference
- Passing an Object Reference to a Prepared Statement

You can define an Oracle object reference to an object stored in an object table. In contrast, you cannot define an object reference for an object value that is stored in a table column.

In SQL, object references (REFs) are 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, it is materialized as an instance of the oracle.sql.REF class and is not strongly typed. So, if you select an EMPLOYEE REF, an oracle.sql.REF object is returned. To find out what kind of REF it really is, use the object’s getTypeName() method. This method returns the object’s SQL type, which in this case would be EMPLOYEE.

An object reference is a primitive SQL type. The steps to access and manipulate object references are similar to the steps you employ for any other primitive SQL type.

**Note:** You cannot have a reference to an array, even though arrays, like objects, are structured types.

JDBC provides support for REFS as any of the following:

- columns in a SELECT list
- IN or OUT bind variables
- attributes in an Oracle8 object
- elements in a collection (array) type object

If you use JPublisher to generate custom Java classes, then it also generates reference classes. These reference classes are extensions of oracle.sql.REF and, unlike the oracle.sql.REF class, are strongly typed. For example, if you define an Oracle object EMPLOYEE, then JPublisher generates an Employee class and an EmployeeRef class.
Retrieving an Object Reference

To demonstrate how to retrieveRefs, the following example first defines an Oracle object type ADDRESS:

```sql
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 three steps, where stmt is a previously defined statement object. The PEOPLE database table is defined earlier in this section:

```java
ResultSet rs = stmt.executeQuery("SELECT col3 FROM PEOPLE");
rs.next();
REF ref = 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:

```java
REF ref = (REF)rs.getObject(1);
```
There is no advantage or disadvantage in using `getObject()` instead of `getREF()`.

**Passing an Object Reference to a Callable Statement**

To retrieve an object reference as an OUT parameter in PL/SQL blocks, do the following to register the bind type for your OUT parameter.

1. Cast your callable statement to an `OracleCallableStatement`:
   ```java
   OracleCallableStatement ocs =
       (OracleCallableStatement) conn.prepareCall("(? = call func())")
   ```

2. Register the OUT parameter with this form of the `registerOutParameter()` method:
   ```java
   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 type code (in this case, `OracleTypes.REF`). The `sql_type_name` is the name of the STRUCT to which this object reference points. For example, if the OUT parameter is a REF to an ADDRESS object (as in the previous section), then ADDRESS is the `sql_type_name` that should be passed in.

3. Execute the call:
   ```java
   ocs.execute()
   ```

**Accessing and Updating Object Values through an Object Reference**

You could then create a Java `Address` object and update a database ADDRESS object through the reference as follows (omitting whatever would be required for the constructor of the `Address` class). This example assumes that you have already retrieved a valid REF object:

```java
Address addr = new Address(...);
ref.setValue(addr);
```

Here, the `setValue()` method updates the database ADDRESS object.
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 preceding example, use a prepared statement to update an address reference based on ROWID, as follows:

```java
PreparedStatement pstmt =
   conn.prepareStatement("update PEOPLE set ADDR_REF = ? where ROWID = ?");
pstmt.setREF(1, addr_ref);
pstmt.setROWID(2, rowid);
```
Working with Arrays

This section has these subsections:

- Retrieving an Array and its Elements
- Passing an Array to a Prepared Statement
- Passing an Array to a Callable Statement
- Using a Type Map to Map Array Elements

The `oracle.sql.ARRAY` class enables you to access and manipulate arrays and their data within a JDBC program. The `oracle.sql.ARRAY` class implements the `oracle.jdbc2.Array` interface.

JDBC provides support for arrays as any of the following:

- columns in a `SELECT` list
- IN or OUT bind variables
- attributes in an Oracle object

---

**Note:** The term `arrays` in JDBC 2.0 is equivalent to `collections` in Oracle terminology.

---

Arrays include `varrays` (variable-length arrays) and nested tables. The methods in the `oracle.sql.ARRAY` class enable you to access and manipulate the array and its data even if it is a `varray` or nested table. That is, you do not have to add any special code when you are accessing a `varray` or nested table. The methods can determine if they are being applied to a `varray` or nested table, and respond by taking the appropriate actions.

Oracle supports only `named` arrays, where you specify a SQL type name to describe a type of array. The SQL type name is assigned to the array when you create it, as in the following SQL syntax:

```sql
CREATE TYPE <sql_type_name> AS <datatype>
```

The array can be either a nested table or a `varray`.

A `varray` is an array of varying size, thus the name "`varray`". A `varray` has an ordered set of data elements. All elements of a given `varray` 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

---

Oracle Extensions 4-87
the **varray**. You must specify a maximum size when you declare the array type. For example:

```sql
CREATE TYPE myNumType AS VARRAY(10) OF NUMBER;
```

This statement defines `myNumType` as a SQL type name that describes a **varray** of `NUMBER`s that can contain no more than 10-elements.

A nested table is an unordered set of data elements, all of the same datatype. It 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:

```sql
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`.

The remainder of this section describes how to access and update array data. For general information about the `oracle.sql.ARRAY` class, including how to manually create array objects, see "Class oracle.sql.ARRAY" on page 4-14. For a complete code example of creating a table with an array column, then manipulating and printing the contents, see "Array Sample" on page 7-16.

## Retrieving an Array and its Elements

When you retrieve an array you get an `oracle.sql.ARRAY` object where each array element can be returned as a materialized Java array object or as a result set object.

You can retrieve a SQL array that has been selected into 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, you can get the data with the `getObject()` method of the `oracle.sql.ResultSet` class, then cast the output to `oracle.sql.ARRAY`.

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 provides versions of these methods that enable you to specify a type map so you can choose how you want your SQL datatypes to map to Java datatypes. Oracle also provides methods that enable you to retrieve all of an array’s elements or a subset of the array (but note, there is no performance advantage in retrieving a subset of an array as opposed to retrieving the entire array).

---

**Note:** Beginning in release 8.1.5, arrays are indexed from 1. In previous releases, arrays were indexed from 0.

---

**getArray() Method:** The `getArray()` method retrieves the element values of the array into a `java.lang.Object[]` array. The elements are converted to the Java types corresponding to the SQL type of the data in the original array.

The `getArray()` materializes the data as an array of `oracle.sql.*` objects and does not use a type map. Oracle also provides a `getArray(map)` method to let you specify a type map and a `getArray(index,count)` method to retrieve a subset of the array.

**getOracleArray() Method:** The `getOracleArray()` method retrieves the element values of the array into a `Datum[]` array. The elements are converted to the `oracle.sql.*` datatype corresponding to the SQL type of the data in the original array.

---

**Note:** The `getOracleArray()` method is an Oracle-specific extension and does not belong to the `oracle.jdbc2.ARRAY JDBC 2.0` interface.

---

The `getOracleArray()` method materializes the data as an array of `oracle.sql.*` objects and does not use the type map. Oracle also provides the `getOracleArray(index,count)`.

**getResultSet() Method:** 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 that you use `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.

**Retrieving All of an Array’s Elements**

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,

```java
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.

Similarly, if you use `getResultSet()` to return an array of primitive datatypes, then the JDBC drivers return a `ResultSet` object that contains, for each element, the index into the array for the element and the element value. For example:

```java
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 `BigDecimal` element value.

**Retrieving Array Elements According to a Type Map**

By default, if 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:

```java
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`.

The `getResultSet(map)` method behaves in a similar manner to `getArray(map)`.

For more information on using type maps with arrays, see "Using a Type Map to Map Array Elements" on page 4-94.

Retrieving a Subset of an Array’s Elements

To retrieve a subset of the array, you can 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:

```java
Object object = arr.getArray(index, count, map);
Object object = arr.getArray(index, count);
```

Similar examples using `getResultSet()` are:

```java
ResultSet rset = arr.getResultSet(index, count, map);
ResultSet rset= arr.getResultSet(index, count);
```

A similar example using `getOracleArray()` is:

```java
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.
Retrieving an Array as an `oracle.sql.Datum`

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 SQL array elements. For example,

```java
Datum arraydata[] = arr.getOracleArray();
```

where `arr` is an `oracle.sql.ARRAY` object. For an example of retrieving an array and its contents, see "Array Sample" on page 7-16.

Example: Getting and Printing an Array of Primitive Datatypes from a Result Set

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 object of type `oracle.sql.ARRAY`.

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.

```java
stmt.execute("CREATE TYPE num_varray AS VARRAY(10) OF NUMBER(12, 2)"");
stmt.execute("CREATE TABLE varray_table (col1 num_varray)"感);
stmt.execute("INSERT INTO varray_table VALUES (num_varray(100, 200))"感);
ResultSet rs = stmt.executeQuery("SELECT * FROM varray_table");
ARRAY my_array = ((OracleResultSet)rs).getARRAY(1);

// return the SQL type names, integer codes,
// and lengths of the columns
System.out.println("Array is of type " + array.getSQLTypeName());
System.out.println("Array element is of type code " + array.getBaseType());
System.out.println("Array is of length " + array.length());

// get Array elements
BigDecimal[] values = (BigDecimal[]) my_array.getArray();
```
for (int i=0; i<values.length; i++)
{
    BigDecimal out_value = (BigDecimal) values[i];
    System.out.println(">> index " + i + " = " + out_value.intValue());
}

Note that if you use getResultSet() to obtain the array, 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));
}

**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):

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 "Class oracle.sql.ARRAY" on page 4-14 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 "Working with Arrays" on page 4-87 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, elements);

   where descriptor is the ArrayDescriptor object previously constructed and elements is a java.lang.Object containing a Java array of the elements. These objects are converted to raw bytes of the appropriate SQL type.
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.

   ```java
   (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.

---

**Note:** You can use arrays as either IN or OUT bind variables.

### Passing an Array to a Callable Statement

To retrieve a collection as an OUT parameter in PL/SQL blocks, do the following to register the bind type for your OUT parameter.

1. Cast your callable statement to an `OracleCallableStatement`:

   ```java
   OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("(? = call func())")
   ```

2. Register the OUT parameter with this form of the `registerOutParameter()` method:

   ```java
   ocs.registerOutParameter(int param_index, int sql_type, string sql_type_name);
   ```

   where `param_index` is the parameter index, `sql_type` is the SQL type code, and `sql_type_name` is the name of the array type. In this case, the `sql_type` is `OracleTypes.ARRAY`.

3. Execute the query:

   ```java
   ocs.executeQuery()
   ```

### Using a Type Map to Map Array Elements

If your array contains Oracle objects, then you can use a type map to associate each object in the array with its 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 the element is returned as an `oracle.sql.STRUCT`.

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 them to the type map if they are not already there. For instructions on how to add entries to an existing type map or how to create a new type map, see “Understanding Type Maps” on page 4-66.

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.

```java
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 select all of the employees belonging to the SALES department as the custom Java object `EmployeeObj`, then you must create a mapping in the type map between the `EMPLOYEE` SQL type and the `EmployeeObj` custom Java 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 that the `getARRAY()` method can retrieve the `EMPLOYEE_LIST` object into the `employeeArray` object.
Working with Arrays

```java
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);"
Additional Oracle Extensions

This section has the following subsections:

- Performance Extensions
- Additional Type Extensions

This section describes Oracle extensions not related to datatypes in the JDBC 2.0 specification. This consists of additional datatype extensions as well as performance extensions.

Performance Extensions

Oracle JDBC drivers support these extensions that improve performance by reducing round trips to the database:

- Prefetching rows 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.

- Batching updates also reduces round trips to the database, saving on the client side a number of updates that are to be made, and then going to the database once to execute all the updates.

- Specifying column types gets around an inefficiency in the usual JDBC protocol for performing and returning the results of queries.

- Suppressing database metadata TABLE_REMARKS columns avoids an expensive outer join operation.

Oracle supports several extensions to connection properties objects to support these performance extensions. The properties object extensions enable you to set the remarksReporting flag and default values for prefetching and update-batching. For more information, see "Oracle Extensions for Connection Properties" on page 4-109.

Note: The prefetching and batch update extensions were designed prior to the announcement of the JDBC 2.0 standard. They do not match JDBC 2.0.
Row Prefetching
Oracle JDBC drivers 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.

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.

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 number 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 and "Oracle Extensions for Connection Properties" on page 4-109 for more information about the `Properties` object and connection properties.

**Example: Row Prefetching** The following example illustrates the row prefetching feature. It assumes you have imported the `oracle.jdbc.driver.*` classes.

```java
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 () );
```

```java
Example 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 () );
```

```java
Example 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) );

stmt.close();

**Database Update Batching**

Oracle JDBC drivers allow you to accumulate inserts and updates of prepared statements at the client and send them to the server in batches, reducing round trips to the server. You might want to do this when you are repeating the same statement with different bind variables.

Normally JDBC makes a round trip to the database to execute a prepared statement whenever the statement’s `executeUpdate()` method is called. The Oracle update-batching feature, however, associates a batch value with each prepared statement object. Oracle JDBC accumulates execution requests for the prepared statement, then automatically passes them all to the database for execution once the batch value is reached.

**Update Batching Limitations** You can use update batching with `CallableStatement` except when the `CallableStatement` has `OUT` parameters. In this case, the driver automatically overrides any previous batch value and resets it to 1.

Do not use the `addBatch()` and `executeBatch()` methods of the JDBC 2.0 `PreparedStatement` interface. These methods are not consistent with the functionality offered by the methods associated with the `OraclePreparedStatement`.

Regardless of the batch value of an Oracle prepared statement, if any of the bind variables of the statement is (or becomes) a streaming type, then JDBC sets the batch value to 1 and sends any queued requests to the database for execution.

JDBC automatically executes the statement’s `sendBatch()` method whenever the connection receives a commit request, the statement receives a close request, or the connection receives a close request.

If you use the form of the `DriverManager.getConnection()` method that takes a `Properties` object as an argument, then you can set the connection’s default batch value in the object. See "Oracle Extensions for Connection Properties" on page 4-109 for more information about `Properties` objects.

The default batch update value is 1.
**Setting Update Batch Value for Individual Statements**  
You can set the batch value for any individual Oracle prepared statement by applying it to the `OraclePreparedStatement` object. The batch value that you set for an individual statement overrides the value set for the connection. You can also set a default batch value that will apply to any Oracle prepared statement in your Oracle connection by applying it to the `OracleConnection` object.

Follow these steps to apply the Oracle batch value feature for a particular prepared statement:

1. Write your prepared statement and specify input values for the first row:
   ```java
   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 `setDefaultExecuteBatch()` method. In this example, the default batch size of the statement is set to 2.
      ```java
      ((OraclePreparedStatement)ps).setDefaultExecuteBatch(2);
      
      If you wish, insert the `getExecuteBatch()` method at any point in the program to check the default batch value for the statement:
      ```java
      System.out.println(" Statement Execute Batch Value "+
                        ((OraclePreparedStatement)ps).getExecuteBatch());
      
      3. If you send an execute update statement to the database at this point, then no data will be sent to the database. Instead, a call to `executeUpdate()` will return 0.
         ```java
         // 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.
            ```java
            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();

**Overriding the Default Batch Update Value**  If you want to execute accumulated statements before the batch value is reached, then use the `sendBatch()` method of the `OraclePreparedStatement` object. For example:

1. Cast your connection to an `OracleConnection` object and apply the `setDefaultExecuteBatch()` method for the connection. This example sets the default batch for all statements in the connection to 50.

   ```java
   ((OracleConnection)conn).setDefaultExecuteBatch (50);
   ```

2. Write your prepared statement and specify input values for the first row as usual, then execute the statement:

   ```java
   PreparedStatement ps =
       conn.prepareStatement("insert into dept values (?, ?, ?)");
   ```

   ```java
   ps.setInt (1, 32);
   ps.setString (2, "Oracle");
   ps.setString (3, "USA");
   ```

   ```java
   System.out.println (ps.executeUpdate());
   ```

   The execute update does not happen at this point. The `ps.executeUpdate()` method returns "0".

3. If you enter a set of input values for a second row and an `executeUpdate()`, the data will still not be sent to the database since the batch default value for the statement is the same as for the connection: 50.

   ```java
   ps.setInt (1, 33);
   ps.setString (2, "Applications");
   ps.setString (3, "Indonesia");
   ```

   ```java
   // this execute does not actually happen at this point
   int rows = ps.executeUpdate();
   ```

   ```java
   System.out.println ("Number of rows updated before calling sendBatch: "
        + rows);
   ```

   Note that the value of `rows` in the `println` statement is "0".
4. If you apply the `sendBatch()` method at this point, then the two previously batched executes will be sent to the database in a single round trip. The `sendBatch()` method also returns the number of updated rows. This property of `sendBatch()` is used by `println` to print the number of updated rows.

```java
// 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();
```

**Setting Update Batch Value for the Connection** You can specify a default batch value for any Oracle prepared statement in your Oracle connection. To do this, set the `setDefaultExecute()` method on the `OracleConnection` object. For example, the following statement sets the default batch value for all prepared statements belonging to the `conn` connection object to 20:

```java
((OracleConnection)conn).setDefaultExecuteBatch(20);
```

Even though this sets the default batch value for all of the prepared statements belonging to the connection, you can override it by calling `setDefaultBatch()` on individual statements.

**Checking Batch Value** The `getExecuteBatch()` method enables you to check the current setting of the default batch value for a specific Oracle prepared statement object or for all of the prepared statements that belong to the Oracle connection. For example:

```java
Integer batch_val = ((OraclePreparedStatement)ps).getExecuteBatch();
```

**OR**

```java
Integer batch_val = ((OracleConnection)conn).getDefaultExecuteBatch();
```

**Example: Update Batching** The following example illustrates how you use the Oracle update batching feature. It assumes you have imported the `oracle.jdbc.driver.*` classes.

```java
Connection conn =
    DriverManager.getConnection("jdbc:oracle:oci8:","scott","tiger");
```
Additional Oracle Extensions

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
ps.close();
Redefining 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.

Redefining Column Types Limitations To use this feature, you must specify a datatype
for each 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 SQLException.

You cannot define column types for objects or object references.

Redefining Column Types for a Query Following these general steps to redefine 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.

Notes:
- Each statement has its own batch count. Only executes on a
particular statement add to the batch count.
- Updates deferred through batching can affect the results of
other queries. In the following example, if the first query is
defered 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";
3. Determine the following for each column of the expected result set:
   - column index (position)
   - code for the type of the expected return data (which can differ from the
column type)
     
     This is according to oracle.jdbc.driver.OracleTypes for
Oracle-specific types, and according to either java.sql.Types or
OracleTypes for standard types (constants for standard types have the
same value in Types and OracleTypes).

4. For each column of the expected result set, invoke the defineColumnType(),
method of your Statement object, passing it these parameters:
   - column index (integer)
   - type code (integer)
     
     Use the static constants of the java.sql.Types class or, for
Oracle-specific types, the static constants of the
oracle.jdbc.driver.OracleTypes class (such as Types.INTEGER,
Types.FLOAT, Types.VARCHAR, OracleTypes.VARCHAR, and
OracleTypes.ROWID).
   - (optionally) maximum field size (integer)

For example, assuming stmt is an Oracle statement, use this syntax:

stmt.defineColumnType(column_index, type);

OR

stmt.defineColumnType(column_index, type, max_size);

Set maximum field size if you do not want to receive the full default length of
the data. Less data than this maximum size will be returned if the maximum
field size is set to a smaller value using the setMaxFieldSize() method of
the standard JDBC Statement class, or if the natural maximum size of the
datatype is smaller. 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.

**Example: Defining Column Types** The following example illustrates the use of this feature. It assumes you have imported the `oracle.jdbc.driver.*` classes.

```java
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 `Types.CHAR` or `Types.VARCHAR`.

Table 4–6 lists the valid column definition arguments you can use in the `defineColumnType()` method.
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.

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.

```java
( (oracle.jdbc.driver.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

### Table 4–6 Valid Column Type Specifications

<table>
<thead>
<tr>
<th>If the column has Oracle SQL type:</th>
<th>You can use <code>defineColumnType()</code> to redefine it as:</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER, VARNUM</td>
<td>BIGINT, TINYINT, SMALLINT, INTEGER, FLOAT, REAL, DOUBLE, NUMERIC, DECIMAL, CHAR, VARCHAR</td>
</tr>
<tr>
<td>CHAR, VARCHAR2</td>
<td>CHAR, VARCHAR</td>
</tr>
<tr>
<td>LONG</td>
<td>CHAR, VARCHAR, LONGVARCHAR</td>
</tr>
<tr>
<td>LONGRAW</td>
<td>LONGVARBINARY, VARBINARY, BINARY</td>
</tr>
<tr>
<td>RAW</td>
<td>VARBINARY, BINARY</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE, TIME, TIMESTAMP, CHAR, VARCHAR</td>
</tr>
<tr>
<td>ROWID</td>
<td>ROWID</td>
</tr>
</tbody>
</table>
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, stand-alone 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 "%".

**Oracle Extensions for Connection Properties**

One of the forms of the DriverManager.getConnection() method enables you to specify a URL and a properties object:

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

where the URL is of the form:

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

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

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

Table 4–7 lists the connection properties that Oracle JDBC drivers support, including the Oracle extensions for defaultRowPrefetch, remarksReporting, and defaultBatchValue.
The following example shows how to use the `java.util.Properties.put()` method to set performance extension options before connection to the database.

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

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

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

Oracle JDBC drivers support the Oracle-specific datatypes ROWID and REF CURSOR, which were introduced in Oracle7 and are not part of the standard JDBC specification.

ROWID is supported as a Java string and REF CURSOR as a JDBC result set.

Oracle ROWID Type

A ROWID is an identification tag that is unique for each row of an Oracle database table. ROWID can be thought of as a virtual column, containing the ID for each row. The oracle.sql.ROWID class is supplied as a wrapper for ROWID SQL data.

ROWIDs provide functionality similar to the java.sql.ResultSet.getCursorName() and java.sql.Statement.setCursorName() JDBC methods, which are not supported by the Oracle implementation.

If you include the ROWID pseudo-column in a query, then you can retrieve the ROWIDs with the ResultSet.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 immediately follows.

Notes:

- The oracle.sql.ROWID class replaces oracle.jdbc.driver.ROWID, which was used in previous releases of Oracle JDBC.
- Refer to the Javadoc for information about features of the ROWID class.

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. So, 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 that is being referenced. "REF CURSOR", then, identifies a
reference to a cursor variable. Since 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 cursor variables.

To create a cursor variable, begin by identifying a user-defined 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 user-defined type
DeptCursorTyp:

depth_cv DeptCursorTyp  -- declare cursor variable
...

A REF CURSOR, then, is a category of datatype rather than a datatype.

Stored procedures can return user-defined types, or 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 CURSORs are materialized as ResultSet objects and can be accessed
like this:
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.

**Example: Accessing REF CURSOR Data** This example shows how to access REF CURSOR data.

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

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

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

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

In the preceding example:

- A CallableStatement object is created by using the prepareCall() method of the connection class.

- The callable statement implements a PL/SQL procedure which returns a REF CURSOR.

- As always, the output parameter of the callable statement must be registered to define its type. The Oracle type code to use for a REF CURSOR is OracleTypes.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 CURSOR Sample" on page 7-14.
Oracle JDBC Notes and Limitations

The following limitations exist in the Oracle JDBC implementation, but all are either insignificant or have easy work-arounds.

**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 4-111.

**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 5-26.

**PL/SQL TABLE, BOOLEAN and RECORD Types**

Oracle JDBC drivers do not support calling arguments or return values of the PL/SQL TABLE, BOOLEAN, or RECORD types. This is a restriction of the OCI layer.

As a work-around for booleans, you can define an additional PL/SQL stored procedure that accepts the BOOLEAN argument as a CHAR or NUMBER and passes it as a BOOLEAN to the original stored procedure. For more information on this topic, see "Boolean Parameters in PL/SQL Stored Procedures" on page 6-7.

**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 $(1 - 10^{-38}) \times 10^{126}$ to full 38-digit precision.
Read-Only Connection
The read-only connection is not supported. There is no Oracle equivalent to the read-only connection.

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 4-108.

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 do not support SQLWarning.

For information on how the Oracle JDBC drivers handle errors, see "Error Messages and JDBC" on page 3-25.

Bind by Name
Bind 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.

```java
PreparedStatement p = conn.prepareStatement("SELECT name FROM EMP
   WHERE id = :EmpId");
p.setInt(1, 314159);
```

The capability 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:

```java
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 since 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.
This chapter describes advanced JDBC topics, including the following:

- Using NLS
- Working with Applets
- JDBC on the Server: the Server Driver
- Embedded SQL92 Syntax
Using NLS

This section contains these subsections:

- **How JDBC Drivers Perform NLS Conversions**
- **NLS Restrictions**

Oracle’s JDBC drivers support NLS (National Language Support). NLS lets 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, the driver provides the support to perform the conversions between the database character set and the client character set.

For more information on NLS, NLS environment variables, and the character sets that Oracle supports, see the *Oracle8i National Language Support Guide*. See the *Oracle8i 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 NLS character set conversion:

- **java.sql.ResultSet** methods `getString()` and `getUnicodeStream()` return values from the database as Java strings and as a stream of Unicode characters, respectively.

- **oracle.sql.CLOB** method `getCharacterStream()` returns the contents of a CLOB as a Unicode stream.

- **oracle.sql.CHAR** methods `getString()`, `toString()`, and `getStringWithReplacement()` convert the following data to strings:
  - `getString()`: converts the sequence of characters represented by the CHAR object to a string and returns a Java String object.
  - `toString()`: identical to `getString()`, but if the character set is not recognized, `toString()` returns a hexadecimal representation of the CHAR data.
  - `getStringWithReplacement()`: 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 NLS Conversions**

The techniques that Oracle’s 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.

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, 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 NLS**

In the case of a JDBC OCI driver installation, note that there is a client-side character set as well as a database character set. The client character set is determined at client-installation time by the value of the NLS_LANG environment variable. The database character set is determined at database creation. The character set used by the client can be different from the character set used by the database on the server.

So, when performing character set conversion, the JDBC OCI driver has to take three factors into consideration:

- 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 the value of NLS_LANG is not specified, or if it is set to 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.

- If the value of NLS_LANG is set to 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 value of NLS_LANG to convert the data from the database character set to UTF-8; the JDBC driver then converts the UTF-8 data to UCS-2.
If your applications or applets use the JDBC Thin driver, then there will not be an Oracle client installation. Because of this, the OCI client conversion routines in C will not be available. In this case, the client conversion routines are different from the JDBC OCI driver.

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.

Notes:

- The driver sets the value of NLS_LANG 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 Oracle8i National Language Support Guide.

JDBC Thin Driver and NLS

If your JDBC code running in the server accesses the database, then the JDBC Server driver performs a character set conversion based on the database character set. The target character set of all Java programs is UCS-2.

The JDBC Server driver supports the ASCII (US7ASCII) and ISO-Latin-1 (WE8ISO8859P1) character sets only.

Note: The OCI and Thin drivers both provide the same transparent support for NLS.

Server Driver and NLS

If your JDBC code running in the server accesses the database, then the JDBC Server driver performs a character set conversion based on the database character set. The target character set of all Java programs is UCS-2.
NLS Restrictions

Data Size Restriction for NLS Conversions

There is a limit to the maximum sizes for CHAR and VARCHAR2 datatypes when used in bind calls. This limitation is necessary to avoid data corruption. This problem happens only with binds (not for defines) and it affects only CHAR and VARCHAR2 datatypes if you are connected to a multi-byte character set database.

The maximum bind lengths are limited in the following way:

CHARs and VARCHAR2s experience character set conversions that could result in an increase in the length of the data in bytes. The ratio between data sizes before and after a conversion is called the NLS Ratio. After conversion, the bind values should not be greater than 4 Kbytes (in Oracle8), or 2 Kbytes (in Oracle7).

For example, when connecting to an Oracle8 server, you cannot bind more than:
- \( \text{min} \left( 2000, \frac{4000}{\text{NLS Ratio}} \right) \) for CHAR types
- \( \frac{4000}{\text{NLS Ratio}} \) for VARCHAR2 types

Table 5–2 contains examples of the NLS Ratio and maximum bind values for some common server character sets.

Table 5–1 New Restricted Maximum Bind Length for Client-Side Drivers

<table>
<thead>
<tr>
<th>Driver</th>
<th>Server Version</th>
<th>Datatype</th>
<th>Old Max Bind Length (bytes)</th>
<th>New Restricted Max Bind Length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thin and OCI</td>
<td>V8</td>
<td>CHAR</td>
<td>2000</td>
<td>( \text{min}(2000, \frac{4000}{\text{NLS Ratio}}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VARCHAR2</td>
<td>4000</td>
<td>( \frac{4000}{\text{NLS Ratio}} )</td>
</tr>
</tbody>
</table>

Note: The Java VM supports only the English (US7ASCII) and ISO-Latin1 (WE8ISO8859P1) character sets.
### Table 5–2  NLS Ratio and Size Limits for Common Server Character Sets (Cont.)

<table>
<thead>
<tr>
<th>Server Character Set</th>
<th>NLS Ratio</th>
<th>Maximum Bind Value on Oracle8 Server (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>US7ASCII</td>
<td>1</td>
<td>4000</td>
</tr>
<tr>
<td>ISO 8859-1 through 10</td>
<td>1</td>
<td>4000</td>
</tr>
<tr>
<td>JA16SJIS</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>JA16EUC</td>
<td>3</td>
<td>1333</td>
</tr>
</tbody>
</table>
Working with Applets

This section describes some of the basics about working with applets that use the JDBC Thin driver. It begins with a simple example of coding a JDBC applet, it then describes what you must do to allow the applet to connect to a database. This includes how to use the Oracle8 Connection Manager or signed applets if you are connecting to a database that is 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.

- Coding Applets
- Connecting an Applet to a Database
- Using Applets with Firewalls
- Packaging Applets
- Specifying an Applet in an HTML Page
- Browser Security and JDK Version Considerations

Coding Applets

Except for importing the JDBC interfaces to access JDBC entry points, you write a JDBC applet like any other Java applet. Depending on whether you are coding your applet for a JDK 1.1.1 browser or a JDK 1.0.2 browser, there are slight differences in the code that you use. In both cases, your applet must use the JDBC Thin driver, which connects to the database with TCP/IP protocol.

If you are targeting a JDK 1.1.1 browser (such as Netscape 4.x or Internet Explorer 4.x), then you must:

- import the java.sql package into your program. The java.sql package contains the standard JDBC 1.22 interfaces and is part of the standard JDK 1.1.1 class library.
- register the driver with the oracle.jdbc.driver.OracleDriver() class and specify the driver name in the connect string as thin.

If you are targeting a JDK 1.0.2 browser (such as Netscape 3.x or Internet Explorer 3.x), then you must:

- import the jdbc.sql package into your program.

The jdbc.sql package is not a part of the standard JDK 1.0.2 class library. It is a separate library that you download as part of the JDBC distribution. The
jdbc.sql package was created because JDK 1.0.2 browsers do not allow packages starting with the string “java” to be downloaded. As a work-around, the java.sql package has been renamed to jdbc.sql. This renamed package is shipped with the Oracle JDBC product.

- register the driver with the oracle.jdbc.dnlddriver.OracleDriver() class and specify the driver name in the connect string as dnldthin.

The following sections illustrate the differences in coding an applet for a JDK 1.1.1 browser compared with a JDK 1.0.2 browser.

- Coding Applets for a JDK 1.1.1 Browser
- Coding Applets for a JDK 1.0.2 Browser

**Coding Applets for a JDK 1.1.1 Browser**

If you are coding an applet for a JDK 1.1.1 browser, then import the JDBC interfaces from the java.sql package and load the Oracle JDBC Thin driver.

```java
import java.sql.*;
public class JdbcApplet extends java.applet.Applet
{
    Connection conn; // Hold the connection to the database
    public void init()
    {
        // Register the driver.
        DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
        // Connect to the database.
        conn = DriverManager.getConnection
        ("jdbc:oracle:thin:scott/tiger@www-aurora.us.oracle.com:1521:orcl");
        ...
    }
}
```

In this example, the connect string contains the username and password, but you can also pass them as arguments to getConnection() after obtaining them from the user. For more information on connecting to the database, see "Opening a Connection to a Database” on page 3-3.

**Coding Applets for a JDK 1.0.2 Browser**

If you are coding an applet for a JDK 1.0.2 browser, then import the JDBC interfaces from the jdbc.sql package, load the driver from the oracle.jdbc.dnlddriver.OracleDriver() class, and use the dnldthin sub-protocol in your connect string:
import jdbc.sql.*;
public class JdbcApplet extends java.applet.Applet
{
    Connection conn; // Hold the connection to the database
    public void init ()
    {
        // Register the driver
        DriverManager.registerDriver (new oracle.jdbc.dnlddriver.OracleDriver());
        // Connect to the database
        conn = DriverManager.getConnection
        {"jdbc:oracle:dnldthin:scott/tiger@www-aurora.us.oracle.com:1521:orcl"};
        ...
    }
}

Connecting an Applet to a Database

This section includes the following subsections:

- Connecting to a Database on the Same Host as the Web Server
- Connecting to a Database on a Different Host
- Using the Oracle8 Connection Manager
- Using Signed Applets

The most common task of an applet using the JDBC driver is to connect to and query a database. Because of applet security restrictions, 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 your applet can connect only to a database that is running on the same host as the web server. In this case, the applet can connect to the database directly; no additional steps are required.

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

- If your web browser supports JDK 1.1.x, then you can use a signed applet to connect to the database directly.
This section begins with describing the most simple case, connecting to a database on the same host from which the applet was downloaded (that is, the same host as the web server). It then describes the two different ways in which you can connect to a database running on a different host.

**Connecting to a Database on the Same Host as the Web Server**

If your database is running on the same host from which the applet was downloaded, then you can connect to the database by specifying it in your applet. You specify the database in the connect string of the `getConnection()` method in the `DriverManager` class.

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 username `scott` with password `tiger`, then use either of the two following connect strings:

**using host:port:sid syntax:**

```java
String connString="jdbc:oracle:thin:@prodHost:1521:ORCL";
conn = DriverManager.getConnection(connString, "scott", "tiger");
```

**using TNS keyword-value syntax:**

```java
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.

**Connecting to a Database on a Different Host**

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 the applet’s security restrictions. You can do this by using either the Oracle8 Connection Manager or signed applets.

**Using the Oracle8 Connection Manager**

Oracle8 Connection Manager is a lightweight, highly-scalable program that can receive Net8 packets and re-transmit them to a different server. To a client running Net8, 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 Net8 packets to an
Oracle server running on a different host.

Figure 5–1 illustrates the relationship between the applet, the Oracle8 Connection
Manager, and the database.

![Figure 5–1 Applet, Connection Manager, and Database Relationship](image)

Using the Oracle8 Connection Manager requires two steps that are described in
these sections:

- **Installing and Running the Oracle8 Connection Manager**
- **Writing the Connect String that Targets the Oracle8 Connection Manager**

**Installing and Running the Oracle8 Connection Manager** You must install the Connection
Manager on the web server host. You install it from the Oracle8 distribution media.
Please refer to the *Net8 Administrator’s Guide* if you need more help to install the
Connection Manager.

On the web server host you must 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. Please refer to the
*Net8 Administrator’s Guide* for more information on the options you can enter in a
`CMAN.ORA` file.

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.
Working with Applets

```java
cman = (ADDRESS_LIST =
    (ADDRESS = (PROTOCOL=TCP)
      (HOST=<web-server-host>)
      (PORT=1610)))

cman_profile = (parameter_list =
    (MAXIMUM_RELAYS=512)
    (LOG_LEVEL=1)
    (TRACING=YES)
    (RELAY_STATISTICS=YES)
    (SHOW_TNS_INFO=YES)
    (USE_ASYNC_CALL=YES)
    (AUTHENTICATION_LEVEL=0)
  )
```

Note that the Java Net8 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.

You can find a description of the options listed in the CMAN.ORA file in the Net8 Administrator’s Guide.

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 situation illustrated in Figure 5–1. The web server on which the Connection Manager is running is on host `webHost` and is listening on port 1610. The database to which you want to connect is running on host `oraHost`, listening on port 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 Net8 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 Net8 Administrator’s Guide for more information about source_route addressing.

**Using Signed Applets**

If your browser supports JDK 1.1.x, (for example, Netscape 4.0), then you can use signed applets. Signed applets can request socket connection privileges to other machines. To set this up, you must:

1. Sign the applet. For information on the steps you must follow to sign an applet, see Sun Microsystem’s Signed Applet Example at:

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:
   ```java
   netscape.security.PrivilegeManager.enablePrivilege("UniversalConnect");
   Connection conn = DriverManager.getConnection(...);
   ```

   For more information on writing applet code that asks for permissions, see Netscape’s Introduction to Capabilities Classes at:

3. You must obtain an object-signing certificate. See Netscape’s Object-Signing Resources page at:

   for information on obtaining and installing a certificate.

   For a complete example of a signed applet that uses the Netscape Capabilities classes, see "Creating Signed Applets" on page 7-31.

**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 requests from 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.

You can solve this problem by using a Net8-compliant firewall and connect strings that comply with the firewall configuration. Net8-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 Net8-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.

The following sections describe these topics:

- How Firewalls Work
- Configuring a Firewall for Applets that use the JDBC Thin Driver
How Firewalls Work

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 connect 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.

Connecting through a firewall requires two steps that are 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 a Net8-compliant firewall.

Java applets do not have access to the local system (that is, they cannot get the hostname locally or environment variables) 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 bogus 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 and not on the hostname.
Working with Applets

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 and SID ORCL, and you are going through a firewall on host fireWallHost, listening on port 1610, then use the following connect string:

```java
Connection conn =
    DriverManager.getConnection ("jdbc:oracle:thin:" +
    "@<firewall-host>:(port=1610)" +
    "@<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:

```java
String connString =
    "jdbc:oracle:thin:@ixta.us.oracle.com:1521:orcl";
conn = DriverManager.getConnection (connString,
    "scott", "tiger");
```

will not work.

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:

```java
String connString =
    "jdbc:oracle:thin:@(description=(address_list=
    (address=(protocol=tcp) (port=1600) (host=fireWallHost))
    (address=(protocol=tcp) (port=1521) (host=oraHost)))
    (source_route=yes))", "scott", "tiger");
```

```java
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 of 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 *Net8 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 need your applet classes files and the JDBC driver classes file (this will be either `classes111.zip` if you are targeting the applet to a browser running JDK 1.1.1, or `classes102.zip` if you are targeting the applet to a browser running JDK 1.0.2).

Follow these steps:

1. Move the JDBC driver classes file `classes111.zip` (or `classes102.zip`) to an empty directory.
2. Unzip the driver classes zip file.

If you are targeting a browser running the JDK 1.0.2, then **DELETE** the packages listed in the left-hand column of the following table. Next, ensure that the packages listed in the right-hand column are present. All of the packages listed in the table are included in the JDBC distribution.

<table>
<thead>
<tr>
<th>DELETE these packages:</th>
<th>Ensure that these packages are present:</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql</td>
<td>jdbc.sql</td>
</tr>
<tr>
<td>java.math</td>
<td>jdbc.math</td>
</tr>
<tr>
<td>oracle.jdbc.driver</td>
<td>oracle.jdbc.dnlldriver</td>
</tr>
<tr>
<td>oracle.jdbc.dbaccess</td>
<td>oracle.jdbc.dnlldbaccess</td>
</tr>
<tr>
<td>oracle.jdbc.oracore</td>
<td>oracle.jdbc.dnlldoracore</td>
</tr>
<tr>
<td>oracle.jdbc.util</td>
<td>oracle.jdbc.dnlldutil</td>
</tr>
</tbody>
</table>
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.
   To target a browser running the JDK 1.1.1, the single zip file should contain:
   ■ the files from classes111.zip
   ■ your applet classes
   ■ 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 entry points, omit this file.

   To target a browser running the JDK 1.0.2, the single zip file should contain:
   ■ the files from classes102.zip (minus the files you deleted in Step 2)
   ■ your applet classes
   ■ the jdbc interface files from the jdbc.sql package in the classes/jdbc/sql directory of the JDBC distribution

   **Note:** If you are targeting your applet at a browser running the JDK 1.0.2, then you must package the applet in a zip file. Browsers running the JDK 1.0.2 do not support .jar files.

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:
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. You 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:

```html
<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:

```html
<APPLET WIDTH=500 HEIGHT=200 CODE=JdbcApplet ARCHIVE=JdbcApplet.zip CODEBASE=Applet_Samples
</APPLET>
```
Working with Applets

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"

then 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) that contains the applet classes and resources the applet needs. Oracle recommends the use of a .zip 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.

---

Browser Security and JDK Version Considerations

The communication between an applet that uses the JDBC Thin driver and the Oracle database happens on top of Java TCP/IP sockets.

In a JDK 1.0.2-based web browser, such as Netscape 3.0, an applet can open sockets only to the host from which it was downloaded. For Oracle8 this means that the applet can only connect to a database running on the same host as the web server. If you want to connect to a database running on a different host, then you must connect through the Oracle8 Connection Manager. For more information, see “Using the Oracle8 Connection Manager” on page 5-10.

In a JDK 1.1.1-based web browser, such as Netscape 4.0, an applet can request socket connection privileges and connect to a database running on a different host from the web server host. In Netscape 4.0 you perform this by signing your applet (that is, writing a signed applet), then opening your connection as follows:
netscape.security.PrivilegeManager.enablePrivilege
("UniversalConnect");
connection = DriverManager.getConnection
("jdbc:oracle:thin:scott/tiger@dlsun511:1721:orcl");

Please refer to your browser documentation for more information on how to work
with signed applets. You can also refer to "Using Signed Applets" on page 5-13.
JDBC on the Server: the Server Driver

This section has the following subsections:

- Connecting to the Database with the Server Driver
- Session and Transaction Context for the Server Driver
- Testing JDBC on the Server
- Server Driver Support for NLS

Any Java program, Enterprise JavaBean (EJB), or Java stored procedure that runs in the database, can use the Server driver to access the SQL engine.

The Server driver is intrinsically tied to the 8.1 database and to the Java VM. The driver runs as part of the same process as the database. It also runs within the default session: this is the same session in which the Java VM was invoked.

The Server 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 Java VM 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 Net8 call to access the SQL engine.

The server-side 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.

Connecting to the Database with the Server Driver

As described in the preceding section, the Server driver runs within a default session. You are already "connected". You can use either the Oracle-specific API defaultConnection() method or the standard Java DriverManager.getConnection() method to access the default connection.

Connecting with defaultConnection()

The defaultConnection() method of the oracle.jdbc.driver.OracleServerDriver class is an Oracle extension and always returns the same connection object. You do not need to include a connect string with the statement. For example:

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

class JDBCConnection {
    public static Connection connect() throws SQLException {
        Connection conn = null;
        try {
            // connect with the Server driver
            OracleDriver ora = new OracleDriver();
            conn = ora.defaultConnection();
        }
    } catch (SQLException e) {
        return conn;
    }
}

Note that there is no `conn.close` statement. You cannot close a default connection
made by the Server driver. Calling `close()` on the connection is just a no-op.

**Connecting with DriverManager.getConnection()**

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 connection, 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.

If you connect to the database with the `DriverManager.getConnection()`
method, then use the connect string `jdbc:oracle:kprb:`. For example:

`DriverManager.getConnection("jdbc:oracle:kprb:");`

Note that you could include a user name and password in the string, but because
you are connecting from the server, they would be ignored.

**Session and Transaction Context for the Server Driver**

The server-side driver operates within a default session and default transaction
context. The default session is the session in which the Java VM 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.

If you run Java application code in the server, then you can manage the transaction (COMMTTs and ROLLBACKs) explicitly.

**Testing JDBC on the Server**

Almost any JDBC program that can run on a client can also run on the server. All of 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 "oci8" to "kprb". For example:

```java
Connection conn =
    DriverManager.getConnection("jdbc:oracle:kprb:@" +
    database, user, password);
```

The advantage of using this method is that you need to 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 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:

```java
Connection conn = new oracle.jdbc.driver.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
    {
        Connection conn = new oracle.jdbc.driver.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.");
    }
}

Server Driver Support for NLS

For a description of how the Server driver handles database character set conversions for Java programs, see "Server Driver and NLS" on page 5-4.

Character Set Conversion of oracle.sql.CHAR Data

The Server driver performs character set conversions for oracle.sql.CHAR in C; this is a different implementation than for the client-side drivers. The client-side drivers perform character set conversions for oracle.sql.CHAR in Java. For more information on the oracle.sql.CHAR class, see "Class oracle.sql.CHAR" on page 4-19.
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 work-arounds.

**Disabling Escape Processing** Escape processing for SQL92 syntax is enabled by default. The JDBC drivers perform escape substitution before sending the SQL code to the database. If you want the driver to use regular Oracle SQL syntax instead of SQL92 syntax, then use this statement:

```java
stmt.setEscapeProcessing(false)
```

---

**Note:** Since prepared statements have usually been parsed prior to making 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:

```sql
{d 'yyyy-mm-dd'}
```

where `yyyy-mm-dd` represents the year, month, and day; for example,

```sql
{d '1998-10-22'}
```

The JDBC drivers will replace this escape clause with the equivalent Oracle representation: "22 OCT 1998".

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.

```java
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 which are supported by the Oracle-specific `oracle.jdbc.driver.OracleDatabaseMetaData` and the standard Java `java.sql.DatabaseMetadata` interfaces:

- `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)`, `HOUR(time)`.

Oracle’s JDBC drivers do not support the function keyword, ‘`fn`’. If you try to use this keyword, for example:

```java
{fn concat ("Oracle", "8i") }
```

you will get the error “Non supported SQL92 token at position xx: fn” when you run your Java application. The work-around is to use Oracle SQL syntax.

For example, instead of using the `fn` keyword in embedded SQL92 syntax:

```java
Statement stmt = conn.createStatement();
stmt.executeUpdate("UPDATE emp SET ename = {fn CONCAT('My', 'Name')}");
```

use Oracle SQL syntax:

```java
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 ' & '}:

```java
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 back slash character (\) as an escape character, you must enter it twice (that is, \\). For example:

```java
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 work-around is to use Oracle outer join syntax:

Instead of:

```java
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:

```java
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 function call syntax shown below:

Calls without a return value:

{ call procedure_name (argument1, argument2, ...) }  

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.driver.OracleSql.parse()` method performs the conversions.

```java
import oracle.jdbc.driver.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 which 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')
```
This chapter describes how to optimize and troubleshoot a JDBC application or applet, including the following topics:

- JDBC and Multi-Threading
- Performance Optimization
- Common Problems
- Basic Debugging Procedures
- Transaction Isolation Levels and the Oracle Server
The Oracle JDBC drivers provide full support for programs that use multiple threads. The following example program uses the default Oracle employee database `emp`. The program creates a number of threads. Each thread opens a connection and sends a query to the database for the contents of the `emp` table. The program then displays the thread and the employee name and employee ID associated with it.

Execute the program by entering:

```
java JdbcMTSample [number_of_threads]
```

on the command line where `number_of_threads` is the number of threads that you want to create. If you do not specify the number of threads, then the program creates 10 by default.

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

public class JdbcMTSample extends Thread
{
    // Set default number of threads to 10
    private static int NUM_OF_THREADS = 10;
    int m_myId;

    static     int c_nextId = 1;
    static  Connection s_conn = null;

    synchronized static int getNextId()
    {
        return c_nextId++;
    }

    public static void main (String args [])
    {
        try
        {
            // Load the JDBC driver //
            DriverManager.registerDriver
            (new oracle.jdbc.driver.OracleDriver());

            // If NoOfThreads is specified, then read it
            if (args.length > 1) {
                System.out.println("Error: Invalid Syntax. ");
                System.out.println("java JdbcMTSample [NoOfThreads]");
            }
```

```
System.exit(0);
}
else if (args.length == 1)
    NUM_OF_THREADS = Integer.parseInt (args[0]);

// 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();
}

// wait for all threads to end
for (int i = 0; i < NUM_OF_THREADS; i++)
{
    threadList[i].join();
}

}
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
        conn = DriverManager.getConnection("jdbc:oracle:oci8:0",

"scott","tiger");

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

// 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));

// Close all the resources
rs.close();
stmt.close();
if (conn != null)
    conn.close();
System.out.println("Thread " + m_myId + " is finished. ");

} catch (Exception e)
{
    System.out.println("Thread " + m_myId + " got Exception: " + e);
e.printStackTrace();
return;
}
}
Performance Optimization

You can significantly enhance the performance of your JDBC programs by using any of these features:

- Disabling Auto-Commit Mode
- Prefetching Rows
- Batching Updates

Disabling Auto-Commit Mode

Auto-commit mode indicates to the database whether to issue an execute and commit after every SQL statement. 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 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 (`setAutoCommit(false)`), then the JDBC driver groups the connection’s SQL statements into transactions that it terminates by either a `commit()` or `rollback()` statement.

Example: Disabling AutoCommit

The following example illustrates loading the driver and connecting to the database. Since 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.

```java
// Load the Oracle JDBC driver
DriverManager.registerDriver(new oracle.jdbc.driver.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();
...

**Prefetching Rows**

Oracle JDBC drivers allow you to set the number of rows to prefetch into the client while the result set is being populated during a query. The default number of rows to prefetch is 10. Prefetching row data into the client reduces the number of round trips to the server. In contrast, standard JDBC fetches the result set one row at a time, where each row requires a round trip to the database.

You can set the row prefetching value for an individual statement or for all statements in your connection. For a description of row prefetching and how to enable it, see "Row Prefetching" on page 4-98.

**Batching Updates**

The Oracle JDBC drivers allow you to accumulate inserts and updates of prepared statements at the client and send them to the server in batches once it reaches a specified batch value. This feature reduces round trips to the server. The default batch value is one.

You can set the batch value for any individual Oracle prepared statement or for all Oracle prepared statements in your Oracle connection. For a description of update batching and how to enable it, see "Database Update Batching" on page 4-100.

**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 of 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, serious memory leaks could occur. You could also run out of cursors in the database. Closing a result set or statement objects releases the corresponding cursor in the database.

Similarly, you must explicitly close `Connection` objects to avoid leaks 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 servers 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 Booleans, 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 integer 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 the number of processes on the server exceeded the limit specified in the initialization file or 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 it.
Basic Debugging Procedures

This section describes four strategies for debugging a JDBC program.

- **Trapping Exceptions**
- **Logging JDBC Calls**
- **Net8 Tracing to Trap Network Events**
- **Using Third Party Tools**

Trapping Exceptions

Most errors that occur in JDBC programs are handled as exceptions. Java provides the `try...catch` statement to catch the exception and the `printStackTrace()` method to print the stack trace.

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

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

To illustrate how the JDBC drivers handle errors, the following incorrect code was intentionally added to the `Employee.java` sample:

```java
// 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 block
    catch(SQLException e){ e.printStackTrace (); }
```

Notice an error was intentionally introduced by changing the column index to 5. When you execute the program you get the following error text:

```java
java.sql.SQLException: Invalid column index
at oracle.jdbc.dbaccess.DBError.check_error(DBError.java:235)
at oracle.jdbc.driver.OracleStatement.prepare_for_new_get(OracleStatement.java:1560)
at oracle.jdbc.driver.OracleStatement.getStringValue(OracleStatement.java:1653)
at oracle.jdbc.driver.OracleResultSet.getString(OracleResultSet.java:175)
at Employee.main(Employee.java:41)
```
Basic Debugging Procedures

For more information on how the JDBC drivers handle errors, and the `SQLException()` and the `printStackTrace()` methods, see "Error Messages and JDBC" on page 3-25.

Logging JDBC Calls

You can use the `java.io.PrintStream.DriverManager.setLogStream()` method to log JDBC calls. This method sets the logging/tracing PrintStream used by the DriverManager and all drivers. Insert the following line at the location in your code where you want to start logging JDBC calls:

```java
DriverManager.setLogStream(System.out);
```

Net8 Tracing to Trap Network Events

You can enable client and server Net8 trace to trap the packets sent over Net8. 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 Net8 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 Net8 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. You can find more information on these parameters in the Net8 Administrator’s Guide.

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**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Turns tracing on/off to a certain specified level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value</td>
<td>0 or OFF</td>
</tr>
<tr>
<td>Available Values</td>
<td>0 or OFF - No trace output</td>
</tr>
<tr>
<td></td>
<td>4 or USER - User trace information</td>
</tr>
<tr>
<td></td>
<td>10 or ADMIN - Administration trace information</td>
</tr>
<tr>
<td></td>
<td>16 or SUPPORT - WorldWide Customer Support trace information</td>
</tr>
<tr>
<td>Example</td>
<td>TRACE_LEVEL_CLIENT=10</td>
</tr>
</tbody>
</table>

**TRACE_DIRECTORY_CLIENT**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Specifies the destination directory of the trace file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value</td>
<td>$ORACLE_HOME/network/trace</td>
</tr>
<tr>
<td>Example</td>
<td>on UNIX: TRACE_DIRECTORY_CLIENT=/oracle/traces</td>
</tr>
<tr>
<td></td>
<td>on Windows NT: TRACE_DIRECTORY_CLIENT=C:\ORACLE\TRACES</td>
</tr>
</tbody>
</table>

**TRACE_FILE_CLIENT**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Specifies the name of the client trace file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value</td>
<td>SQLNET.TRC</td>
</tr>
<tr>
<td>Example</td>
<td>TRACE_FILE_CLIENT=cli_Connection1.trc</td>
</tr>
</tbody>
</table>

**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.

Using Third Party Tools

You can use tools such as JDBC Spy and JDBC Test from Intersolv to troubleshoot at the JDBC API level. These tools are similar to ODBC Spy and ODBC Test.

Transaction Isolation Levels and the Oracle Server

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.driver.OracleConnection class 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.
This chapter presents sample applications that highlight advanced JDBC features and Oracle extensions, including the following topics:

- Sample Applications for Basic JDBC Features
- Sample Applications for JDBC 2.0-Compliant Oracle Extensions
- Sample Applications for Other Oracle Extensions
- Creating Customized Java Classes for Oracle Objects
- Creating Signed Applets
- JDBC versus SQLJ Sample Code
Sample Applications for Basic JDBC Features

This section contains code samples that demonstrate basic JDBC features.

Streaming Data

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 using the JDBC OCI driver for connecting to a database, and inserting and fetching LONG data using Java streams.

```java
import java.sql.*;                                           // line 1
import java.io.*;

class StreamExample
{
    public static void main (String args [])
        throws SQLException, IOException
    {
        // Load the driver
        DriverManager.registerDriver(new oracle.jdbc.driver.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);                             // line 18

        // 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                                     // line 34
    }
}
```
stmt.execute ("create table streamexample (NAME varchar2 (256),
    DATA long");

File file = new File ("StreamExample.java");            // line 37
InputStream is = new FileInputStream ("StreamExample.java");
PreparedStatement pstmt =
    conn.prepareStatement ("insert into streamexample (name, data)
        values (?, ?)");
pstmt.setString (1, "StreamExample");
pstmt.setAsciiStream (2, is, (int)file.length ());
pstmt.execute ();                                      // line 44

// 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                                    // line 51
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 ();                                          // line 66
}

Lines 1-18: Import the necessary classes. Load the JDBC OCI driver with the
DriverManager.registerDriver() method. Connect to the database with the
getConnection(), as user scott with password tiger. Use the database URL
jdbc:oracle:oci8:8. You can optionally enter a database name after the @ symbol. Disable AUTOCOMMIT to enhance performance. If you do not, the driver
will issue execute and commit commands after every SQL statement.
Line 34: Create a table STREAMEXAMPLE with a NAME column of type VARCHAR and a DATA column of type LONG.

Lines 37-44: Insert the contents of the StreamExample.java into the table. To do this, create an input stream object for the Java file. Then, prepare a statement to insert character data into the NAME column and the stream data into the DATA column. Insert the NAME data with the setString(); insert the stream data with setAsciiStream().

Line 46: Query the table to get the contents of the DATA column into a result set.

Line 51-66: Get the data from the first row of the result set into the InputStream object gif_data. Create a FileOutputStream to write to the specified file object. Then, read the contents of the gif stream and write it to the file example.out.

Sample Applications for JDBC 2.0-Compliant Oracle Extensions

This section contains sample code for the following Oracle extensions:

- LOB Sample
- BFILE Sample

LOB Sample

This sample demonstrates basic support for LOBs in the OCI 8 driver. 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 LOBs" on page 4-45.

Except for some changes to the comments, the following sample is similar to the LobExample.java program in the Demo/samples/oci8/object-samples directory.

```java
import java.sql.*;                                   // line 1
import java.io.*;
import java.util.*;

// Importing the Oracle Jdbc driver package
// makes the code more readable
import oracle.jdbc.driver.*;

// Import this to get CLOB and BLOB classes
import oracle.sql.*;
```
public class NewLobExample1
{
    public static void main (String args [])
        throws Exception
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.driver.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 auto commit is off
        conn.setAutoCommit (false);                             // line 26

        // 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 but we gleefully ignore it
        }                                                       // line 38

        // Create a table containing a BLOB and a CLOB             line 40
        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',
                     '010101010101010101010101010101', 'onetwothreefour');
        stmt.execute ("INSERT INTO basic_lob_table VALUES ('two',
                     '0202020202020202020202020202', 'twothreefourfivesix');

        System.out.println ("Dumping lobs");

        // Select the lobs
        ResultSet rset = stmt.executeQuery ("SELECT * FROM basic_lob_table");
        while (rset.next ()

Sample Applications  7-5
{ // Get the lobs
    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);
}

System.out.println("Dumping lobs again");

rset = stmt.executeQuery("SELECT * FROM basic_lob_table");
while (rset.next ())
{
    // Get the lobs
    BLOB blob = ((OracleResultSet)rset).getBLOB (2);
    CLOB clob = ((OracleResultSet)rset).getCLOB (3);

    // Print the lob contents
    dumpBlob (conn, blob);
    dumpClob (conn, clob);
}

// 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)
    {
    // line 68
    // line 82
    // line 91
    // line 98
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();
} // line 108

// 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();
} // line 135

// 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);      // line 147

    i += chunk;
    if (length - i < chunk)
        chunk = (int) length - i;
}
outstream.close();                                    // line 154

// Utility function to write data to a Blob
static void fillBlob (Connection conn, BLOB blob, long length)
    throws Exception
{
    OutputStream outstream = blob.getBinaryOutputStream();
    int i = 0;
    byte [] data = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };          // line 165
    int chunk = data.length;

    while (i < length)
    {
        data [0] = (byte)i;
        outstream.write(data, 0, chunk);

        i += chunk;
        if (length - i < chunk)
            chunk = (int) length - i;
    }
    outstream.close();}
}                                                                   // line 175

Lines 1-26: Import the necessary java.* and oracle.* classes. Register the driver with the DriverManager.registerDriver() method and connect to the database with DriverManager.getConnection(). Use the database URL jdbc:oracle:oci8:@ and connect as user scott with password tiger. You can optionally enter a database name following the @ symbol.
Use `setAutoCommit(false)` to disable the AUTOCOMMIT feature and enhance performance. If you do not, the driver will issue execute and commit commands after every SQL statement.

**Lines 27-38:** Create a statement object. Drop any pre-existing table named `basic_lob_table`. Then, create a new `basic_lob_table` directory to store the LOBs in-line.

**Lines 40-49:** Use SQL statements to create a table with three columns: a column to store the row number as a `VARCHAR2`, a `BLOB` column, and a `CLOB` column. Then insert data into two rows of the table.

**Lines 50-68:** `SELECT` the contents of the table into a result set.

Retrieve the LOBs. The `getBLOB()` and `getCLOB()` methods return locators to the LOB data; to retrieve the LOB contents, you must write additional code (which is defined later in this program). To use the `getBLOB()` and `getCLOB()` methods, cast the result set to an `OracleResultSet` object. Then call the "dump" functions to display the contents of the LOBs, and the "fill" functions to change the contents of the LOBs. The `dump` and `fill` functions are defined later in this program.

**Lines 69-82:** Display the LOBs again, after their contents have been changed. `SELECT` the contents of the table into a result set, and then apply the `dump` functions. The `dump` functions are defined later in this program.

**Lines 84-108:** Define the utility function `dumpClob` to display the contents of a `CLOB`. Read the `CLOB` contents as a character stream. Use the `getCharacterStream()` method to get a `READER` stream object. Set up the temporary character array to read the character data in 10-character chunks.

Set up a loop to read and display the contents of the `CLOB`. The length of the `CLOB` is displayed as well. Close the input stream when you are done.

**Lines 110-135:** Define the utility function `dumpBlob` to display the contents of a `BLOB`. Read the `BLOB` contents as a binary stream. Use the `getBinaryStream()` method to get an `InputStream` stream object. Set up the temporary byte array to read the binary data in 10-byte chunks.

Set up a loop to read and display the contents of the `BLOB`. The length of the `BLOB` is displayed as well. Close the input stream when you are done.

**Lines 136-154:** Define the utility function `fillClob` to write data to a `CLOB`. The `fillClob` function needs the `CLOB` locator and the length of the `CLOB`. To write to
the CLOB, use the getCharacterOutputStream() method to get a WRITER object.

Set up a loop to write an index value and part of the string Hello World to the CLOB. Close the WRITER stream when you are done.

Lines 156-175: Define the utility function fillBlob to write data to a BLOB. The fillBlob function needs the BLOB locator and the length of the BLOB. To write to the BLOB, use the getBinaryOutputStream() method to get an OutputStream object.

Define the byte array of data that you want to write to the BLOB. The while loop causes a variation of the data to be written to the BLOB. Close the OutputStream object when you are done.

BFILE Sample

This sample demonstrates basic BFILE support in the OCI 8 driver. It illustrates filling a table with BFILES and includes a utility for dumping the contents of a BFILE. For more information on BFILES, see "Working with LOBs" on page 4-45.

Except for some changes to the comments, the following sample is similar to the FileExample.java program in the Demo/samples/oci8/object-samples directory.

```java
import java.sql.*;                                           // line 1
import java.io.*;
import java.util.*;
//including this import makes the code easier to read
import oracle.jdbc.driver.*/;

// needed for new BFILE class
import oracle.sql.*;                                        // line 10

public class NewFileExample1
{
    public static void main (String args [])
        throws Exception
    {
        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

        // Connect to the database
        // You can put a database name after the @ sign in the connection URL.
```
The example creates a DIRECTORY and you have to be connected as "system" to be able to run the test. If 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.

```java
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 '/temp/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 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);
}

// 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: ");                 // line 77
    bfile.openFile();
    System.out.println ("File open: " + bfile.isFileOpen());

    long length = bfile.length();
    System.out.println ("File 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 ");
    }
    for (int i=0; i<length; i++)
    {
        System.out.print(buffer[i] + " ");
    }
}                                                      // line 108
// Close input stream
instream.close();

// close file handler
bfile.closeFile();
}

Lines 1-32: Import the necessary java.* and oracle.* classes. Register the driver with the DriverManager.registerDriver() method and connect to the database with the getConnection() method. Use the database URL jdbc:oracle:oci8: and connect as user system with password manager. You can optionally enter a database name following the @ symbol.

Use setAutoCommit(false) to disable the AUTOCOMMIT feature and enhance performance. If you do not, the driver will issue execute and commit commands after every SQL statement.

Lines 33-44: Create a statement object. Drop any pre-existing directory named TEST_DIR. Then, create a new TEST_DIR directory to store the BFILE. You or your System Administrator can use whatever file name you wish.

Lines 46-53: Drop any pre-existing table named test_dir_table.

Lines 55-63: Create and populate a table with files. Use SQL statements to create a table, test_dir_table, with two columns: one column to indicate the row number as a VARCHAR2 (for example, "one" or "two"), and one column to hold the BFILE locator.

Use SQL statements to insert some data into the table. For the first row, insert a row number in the first column, and use the BFILENAME keyword to insert a BFILE, file1, located in TEST_DIR, in the second column. Do the same thing for the second row.

Lines 64-75: SELECT the contents of the table into a result set. Set up a loop to retrieve the contents of the table. Use getString() to retrieve the row number data, and use getBFILE() to retrieve the BFILE locator. Since BFILE is an Oracle-specific datatype, and getBFILE() is an Oracle extension, cast the result set object to an OracleResultSet object.

Use the dumpBfile() method (defined later in the program) to display the BFILE contents and various statistics about the BFILE.
Sample Applications for Other Oracle Extensions

Line 77: Define the dumpBfile() method to display the BFILE contents and various statistics about the BFILE. The dumpBfile() method takes the BFILE locator as input.

Lines 80-83: Use the getName(), fileExists(), and isFileOpen() methods to return the name of the BFILE, and whether the BFILE exists and is open. Note that the BFILE does not have to be open to apply these methods to it.

Lines 84-108: Read and display the BFILE contents. First open the BFILE. You can read the BFILE contents as a binary stream. Use the getBinaryStream() method to get an input stream object. Determine the size of the “chunk” in which the stream will read the BFILE data, and set up the temporary byte array to store the data.

Set up a loop to read and display the contents of the BFILE. The length of the BFILE is displayed as well.

Lines 110-115: When you are finished, close the input stream and the BFILE.

Sample Applications for Other Oracle Extensions

This section contains sample code for these Oracle extensions:

- REF CURSOR Sample
- Array Sample

REF CURSOR Sample

Following is a complete sample program that uses JDBC to create a stored package in the data server and uses a get on the REF CURSOR type category to obtain the results of a query. For more information on REF CURSORS, see "Oracle REF CURSOR Type Category" on page 4-112.

Except for some changes to the comments, the following sample is similar to the RefCursorExample.java program in the Demo/samples/oci8/object-samples directory.

import java.sql.*; // line 1
import java.io.*;
import oracle.jdbc.driver.*;

class RefCursorExample
{
    public static void main(String args[]) throws SQLException
    
    7-14 JDBC Developer’s Guide and Reference
{  
    // Load the driver.
    DriverManager.registerDriver(new oracle.jdbc.driver.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 who all the sales people are.
    call.registerOutParameter(1, OracleTypes.CURSOR);
    call.setString(2, "SALESMAN");
    call.execute();
    ResultSet rset = (ResultSet)call.getObject(1);

    // Output the information in the cursor.
    while (rset.next())
        System.out.println(rset.getString("ENAME"));

    // 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; " +
}
import java.sql.*;                                           // line 1
import oracle.sql.*;
import oracle.jdbc.oracore.Util;
import oracle.jdbc.driver.*;
import java.math.BigDecimal;

"end java_cursor;")};                     // line 53

Lines 1-16: Import the necessary java.* and oracle.* classes. Register the driver with the DriverManager.registerDriver() method and connect to the database with the getConnection() method. Use the database URL jdbc:oracle:oci8:@ and connect as user scott with password tiger. You can optionally enter a database name following the @ symbol.

Lines 18-29: Prepare a callable statement to the job_listing function of the java_refcursor PL/SQL procedure. The callable statement returns a cursor to the rows of information where job=SALESMAN. Register OracleTypes.CURSOR as the output parameter. The setObject() method passes the value SALESMAN to the callable statement. After the callable statement is executed, the result set contains a cursor to the rows of the table where job=SALESMAN.

Lines 30-33: Iterate through the result set and print the employee name part of the employee object.

Lines 40-45: Define the package header for the java_refcursor package. The package header defines the return types and function signatures.

Lines 46-53: Define the package body for the java_refcursor package. The package body defines the implementation which selects rows based on the value for job.

Array Sample

Following is a complete sample program that 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 "Working with Arrays" on page 4-87.

Except for some changes to the comments, the following sample is similar to the ArrayExample.java program in the Demo/samples/oci8/object-samples directory.
public class ArrayExample
{
    public static void main (String args[])
    throws Exception
    {
        // Register the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

        // Connect to the database
        // You need to put your database name after the @ symbol in
        // the connection URL.
        //
        // The sample retrieves an varray of type "NUM_VARRAY" and
        // materializes the object as an object of type ARRAY.
        // A new ARRAY is then inserted into the database.

        // Please replace hostname, port_number and sid_name with
        // the appropriate values
        Connection conn =
            DriverManager.getConnection
            ("jdbc:oracle:oci8:@(description=(address=(host=hostname)(protocol=tcp)(port=port_number))(connect_data=(sid=sid_name)))", "scott", "tiger");

        // It’s faster when auto commit is off
        conn.setLines (false);                                  // line 32

        // Create a Statement
        Statement stmt = conn.createStatement ();               // line 35
        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
        }                                                       // line 47
        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");
showResultSet (rs); // line 54

//now insert a new row

// create a new ARRAY object
int elements[] = { 300, 400, 500, 600 }; // line 59
ArrayDescriptor desc = ArrayDescriptor.createDescriptor("NUM_VARRAY", conn);
ARRAY newArray = new ARRAY(desc, conn, elements); // line 62
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);
} // line 70

public static void showResultSet (ResultSet rs) // line 72
    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 type code "
            + array.getBaseType());
        System.out.println ("Array is of length " + array.length()); // line 86

        // get Array elements
        BigDecimal[] values = (BigDecimal[]) array.getArray();

        for (int i=0; i<values.length; i++)
        {
            BigDecimal value = values[i];
            System.out.println(">> index " + i + " = " + value.intValue());
        }
    }
} //
Lines 1-32: Import the necessary java.* and oracle.* classes. Register the driver with the DriverManager.registerDriver() method and connect to the database with the getConnection() method. This example of getConnection() uses Net8 name-value pairs to specify the host as hostname, protocol as tcp, port as 1521, sid as orcl, user as scott and password as tiger.

Use setAutoCommit(false) to disable the AUTOCOMMIT feature and enhance performance. If you do not, the driver will issue execute and commit commands after every SQL statement.

Lines 35-47: Create a Statement object and delete any previously defined tables or types named varray_table or num_varray.

Lines 49-54: Create the type num_varray as a varray containing NUMBER data. Create a 1-column table, varray_table, to contain the num_varray type data. Insert into the table two rows of data. The values 100 and 200 are both of type num_varray. Use the showResultSet() method (defined later in the program) to display information about the arrays contained in the table.

Lines 59-61: First, define an array of integer elements to insert into the varray_table. Next, create an array descriptor object that will be used to create new ARRAY objects. To create an array descriptor object, pass the SQL type name of the array type (NUM_ARRAY) and the connection object to the createDescriptor() method. Then create the new array object by passing to it the array descriptor, the connection object, and the array of integer elements.

Lines 63-70: Prepare a statement to insert the new array object into varray_table. Cast the prepared statement object to an OraclePreparedStatement object to take advantage of the setARRAY() method.

To retrieve the array contents of the table, write and execute a SQL SELECT statement. Again, use the showResultSet() method (defined later in the program) to display information about the arrays contained in the table.

Lines 72-85: Define the showResultSet() method. This method loops through a result set and returns information about the arrays it contains. This method uses the result set getARRAY() method to return an array into an oracle.sql.ARRAY object. To do this, cast the result set to an OracleResultSet object. Once you have the ARRAY object, you can apply Oracle extensions getSQLTypeName(),
getBaseType(), as well as length(), to return and display the SQL type name of the array, the SQL type code of the array elements, and the array length.

**Lines 87-97:** You can access the varray elements by using the ARRAY object’s getArray() method. Since the varray contains SQL numbers, cast the result of getArray() to a java.math.BigDecimal array. Then, iterate through the value array and pull out individual elements.

### Creating Customized Java Classes for Oracle Objects

This section contains the following subsections:

- SQLData Sample
- CustomDatum Sample

This section contains examples of the code you must provide to create custom Java classes for Oracle objects. You create the custom classes by implementing either the SQLData or CustomDatum interface. These interfaces provide a way to create and populate the custom Java class for the Oracle object and its attributes.

Although both SQLData and CustomDatum both populate a Java object from a SQL object, the CustomDatum interface is far more powerful. In addition to letting you populate Java objects, CustomDatum lets you materialize objects from SQL types that are not necessarily objects. Thus, you can create a CustomDatum object from any datatype found in an Oracle database. This is particularly useful in the case of RAW data that can be a serialized object.

The SQLData interface is a JDBC standard. For more information on this interface, see "Understanding the SQLData Interface" on page 4-69.

The CustomDatum interface is provided by Oracle. You can write your own code to create custom Java classes that implement this interface, but you will find it easier to let the Oracle utility JPublisher create the custom classes for you. The custom classes created by JPublisher implement the CustomDatum interface.

For more information on the CustomDatum interface, see "Understanding the CustomDatum Interface" on page 4-75. See the Oracle8i JPublisher User’s Guide for more information on the JPublisher utility.

### SQLData Sample

This section contains a code sample that illustrates how you can create a custom Java type to correspond to a given SQL type. It then demonstrates how you can use
the custom Java class in the context of a sample program. The sample also contains the code to map the SQL type to the custom Java type.

Creating the SQL Object Definition
Following is the SQL definition of an EMPLOYEE object. The object has two attributes: a string EmpName (employee name) attribute and an integer EmpNo (employee number) attribute.

```
-- SQL definition
CREATE TYPE EMPLOYEE AS OBJECT
(
    EmpName VARCHAR2(50),
    EmpNo   INTEGER,
);
```

Creating the Custom Java Class
The following program implements the custom Java class EmployeeObj to correspond to the SQL type EMPLOYEE. Notice that the implementation 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.

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

public class EmployeeObj implements SQLData {
    private String sql_type;

    public EmployeeObj();
    {
    }
    // line 14

    public EmployeeObj (String sql_type, String empName, int empNo)
    {
        this.sql_type = sql_type;
        this.empName = empName;
        this.empNo = empNo;
    }
```
Creating Customized Java Classes for Oracle Objects

import java.*; import oracle.*;

public class EmployeeObj implements SQLData {

    private String sql_type;
    private String empName;
    private int empNo;

    // define a get method to return the SQL type of the object
    public String getSQLTypeName() throws SQLException {
        return sql_type;
    }

    // define the required readSQL() method
    public void readSQL(SQLInput stream, String typeName) throws SQLException {
        sql_type = typeName;
        empName = stream.readString();
        empNo = stream.readInt();
    }

    // define the required writeSQL() method
    public void writeSQL(SQLOutput stream) throws SQLException {
        stream.writeString(empName);
        stream.writeInt(empNo);
    }

    // Lines 1-14: Import the needed java.* and oracle.* packages. Define the custom Java class EmployeeObj to implement the SQLData interface. EmployeeObj is the class to which you will later map your EMPLOYEE SQL object type. The EmployeeObj object has three attributes: a SQL type name, an employee name, and an employee number. The SQL type name is a Java string that represents the fully qualified SQL type name (schema.sql_type_name) of the Oracle object that the custom Java class represents.

    // Lines 24-28: Define a getSqlType() method to return the SQL type of the custom Java object.

    // Lines 30-38: Define a readSQL() method as required by the definition of the SQLData interface. The readSQL() method takes a stream SQLInput object and the SQL type name of the object data that it is reading.
Lines 39-45: Define a `writeSQL()` method as required by the definition of the `SQLData` interface. The `writeSQL()` method takes a stream `SQLOutput` object and the SQL type name of the object data that it is reading.

Using the Custom Java Class

After you create your `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.

Except for some changes to the comments, the following sample is similar to the `SQLDataExample.java` program in the `Demo/samples/oci8/object-samples` directory.

```java
import java.sql.*;                                           // line 1
import oracle.jdbc.driver.*;
import oracle.sql.*;
import java.math.BigDecimal;
import java.util.Dictionary;

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

        // Connect to the database
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        OracleConnection conn = (OracleConnection)
            DriverManager.getConnection("jdbc:oracle:oci8:@", "scott", "tiger");        // line 16

        // in the type map, add the mapping of EMPLOYEE SQL     // line 18
        // type to the EmployeeObj custom Java type
        Dictionary map = conn.getTypeMap();
        map.put("EMPLOYEE", Class.forName("EmployeeObj")); // line 21

        // Create a Statement                                      line 23
        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                                    // line 35
stmt.executeUpdate("CREATE TYPE EMPLOYEE AS OBJECT(EmpName VARCHAR2(50),
    EmpNo INTEGER)");
stmt.executeUpdate("CREATE TABLE EMPLOYEE_TABLE (ATTR1 EMPLOYEE)");
stmt.executeUpdate("INSERT INTO EMPLOYEE_TABLE VALUES (EMPLOYEE('Susan Smith',
    123))");                                          // line 40

// Create a SQLData object EmployeeObj in the SCOTT schema
EmployeeObj e = new EmployeeObj("SCOTT.EMPLOYEE", "George Jones", 456);

// Insert the SQLData object EmployeeObj into the database       // line 45
PreparedStatement pstmt
    = conn.prepareStatement("INSERT INTO employee_table VALUES (?)");
pstmt.setObject(1, e, OracleTypes.STRUCT);
pstmt.execute();
System.out.println("insert done");
pstmt.close();                                          // line 52

// Select the contents of the employee_table                      // line 54
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)
s.executeQuery("SELECT * FROM employee_table");       // line 57

// print the contents of the table                               // line 59
while(rs.next())
{
    EmployeeObj ee = (EmployeeObj) rs.getObject(1);
    System.out.println("EmpName: " + ee.empName + " EmpNo: " + ee.empNo);        // line 64
}

// close the result set, statement, and connection               // line 66
rs.close();
s.close();
if (conn != null)
{
    conn.close();                                         // line 72
}
}
Import needed `java.*` and `oracle.*` packages. Register the driver with the `DriverManager.registerDriver()` method and connect to the database with the `getConnection()` method. Use the database URL `jdbc:oracle:oci8:@` and connect as user `scott` with password `tiger`. You can optionally enter a database name following the `@` symbol.

Lines 18-21: Use the `getTypeMap()` method to get the type map associated with this connection. Use the map object’s `put()` method to add the mapping of the SQL `EMPLOYEE` object to the `EmployeeObj` custom Java type.

Lines 23-33: Create a statement object and drop any pre-existing tables and types named `EMPLOYEE_TABLE` and `EMPLOYEE`.

Lines 35-40: Use SQL statements to:
- create an `EMPLOYEE` object with employee name and employee number attributes
- create a table of employee objects (`EMPLOYEE_TABLE`) having a single `EMPLOYEE` column
- insert initial data values into the table

Lines 42, 43: Create a new `EmployeeObj` object (which is a `SQLData` object). Identify the schema name (`SCOTT`), SQL type name (`EMPLOYEE`), an employee name (`George Jones`) and an employee number (456). Note that the schema name is the same as the user name in the `getConnection()` call. If you change the user name, you must also change the schema name.

Lines 45-52: Prepare a statement to insert the new `EMPLOYEE` object into the employee table. The `setObject()` method indicates that the object will be inserted into the first index position and that the underlying type of the `EMPLOYEE` object is `oracle.sql.STRUCT`.

Lines 54-57: Select the contents of the `EMPLOYEE_TABLE`. Cast the results to an `OracleResultSet` so that you can retrieve the custom Java object data from it.

Lines 59-62: Iterate through the result set, getting the contents of the `EMPLOYEE` objects and printing the employee names and employee numbers.

Lines 66-72: Close the result set, statement, and connection objects.
Creating Customized Java Classes for Oracle Objects

CustomDatum Sample

This section describes a Java class, written by a user, that implements the CustomDatum and CustomDatumFactory interfaces. The custom Java class of type CustomDatum has a static getFactory() method that returns a CustomDatumFactory object. The JDBC driver uses the CustomDatumFactory 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 CustomDatum and CustomDatumFactory interfaces.

The following example illustrates a Java class definition that can be written by a user, given the SQL definition of an EMPLOYEE object.

SQL Definition of EMPLOYEE Object

The following SQL code defines the EMPLOYEE object. The EMPLOYEE object consists of the employee’s name (EmpName) and the employee’s associated number (EmpNo).

```sql
create type EMPLOYEE as object
    (EmpName VARCHAR2(50),
     EmpNo   INTEGER);
```

Java Class Definitions for a Custom Java Object

Below are the contents of the Employee.java file.

```java
import java.math.BigDecimal;
import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.sql.CustomDatum;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.sql.StructDescriptor;

public class Employee implements CustomDatum, CustomDatumFactory // line 10
{
    static final Employee _employeeFactory = new Employee(null, null); //line 13

    public static CustomDatumFactory getFactory()
    {
```
return _employeeFactory;
}                                      // line 18

/* constructor */                                 // line 20
public Employee(String empName, BigDecimal empNo)
{
    this.empName = empName;
    this.empNo = empNo;
}                                          // line 25

/* CustomDatum interface */                     // line 27
public Datum toDatum(OracleConnection c) throws SQLException
{
    StructDescriptor sd =
        StructDescriptor.createDescriptor("SCOTT.EMPLOYEE", c);
    Object [] attributes = { empName, empNo };

    return new STRUCT(sd, c, attributes);
}                                          // line 36

/* CustomDatumFactory interface */             // line 38
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
    if (d == null) return null;
    System.out.println(d);
    Object [] attributes = ((STRUCT) d).getAttributes();

    return new Employee((String) attributes[0],
                           (BigDecimal) attributes[1]);  // line 49
}

/* fields */                                  
public String empName;
public BigDecimal empNo;
}

**Line 10:** As required, the `Employee` class implements the `CustomDatum` and `CustomDatumFactory` interfaces.

**Lines 13-18:** JPublisher defines a `_employeeFactory` object of class `Employee`, which will be returned by the `getFactory()` method and used to create new
Employee objects. The getFactory() method returns an empty Employee object that you can use to create new Employee objects.

**Lines 20-25:** JPublisher defines the Employee Java class to correspond to the SQL EMPLOYEE object. JPublisher creates the Employee class with two attributes: an employee name of type java.lang.String and an employee number of type java.math.BigDecimal.

**Lines 27-36:** The toDatum() method of the CustomDatum interface transforms the EMPLOYEE SQL data into oracle.sql.* representation. To do this, toDatum() uses:

- a STRUCT descriptor that takes the schema name, the SQL object or "type" name, and the connection object as arguments
- an object array that stores the values of the object’s employee name and employee number attributes

The toDatum() returns a STRUCT containing the STRUCT descriptor, the connection object and the object attributes into an oracle.sql.Datum.

**Lines 38-49:** The CustomDatumFactory interface specifies a create() method that is analogous to the constructor of your Employee custom Java class. The create() method takes the Datum object and the SQL type code of the Datum object and returns a CustomDatum instance.

According to the definition, the create() method returns null if the value of the Datum object is null. Otherwise, it returns an instance of the Employee object with the employee name and employee number attributes.

**Custom Java Class Usage Example**

This code snippet presents a simple example of how you can use the Employee class that you created with 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.

Except for some changes to the comments, the following sample is similar to the CustomDatumExample.java program in the Demo/samples/oci8/object-samples directory.

```java
import java.sql.*;               // line 1
import oracle.jdbc.driver.*;
import oracle.sql.*;
import java.math.BigDecimal;
```
public class CustomDatumExample
{
    public static void main(String args []) throws Exception
    {

        // Connect
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver ());
        OracleConnection conn = (OracleConnection)
                DriverManager.getConnection("jdbc:oracle:oci8:8", 
                    "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))");

        // 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
    }
}

Sample Applications  7-29
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)
    s.executeQuery("SELECT * FROM employee_table");

while(rs.next()) // line 54
{
    Employee ee = (Employee) rs.getCustomDatum(1, Employee.getFactory());
    System.out.println("EmpName: " + ee.empName + " EmpNo: " + ee.empNo);
} // line 58
rs.close();
s.close();

if (conn != null)
{
    conn.close();
}
}

Lines 1-16: Import needed java.* and oracle.* packages. Register the driver with the DriverManager.registerDriver() method and connect to the database with the getConnection() method. Use the database URL jdbc:oracle:oci8:@ and connect as user system with password manager. You can optionally enter a database name following the @ symbol.

Lines 18-28: Create a statement object and drop any pre-existing tables and types named EMPLOYEE_TABLE and EMPLOYEE.

Lines 30-35: Use SQL statements to:
- create an Employee object with employee name and employee number attributes
- create a table of employee objects having a single EMPLOYEE column
- insert initial data values into the table

Lines 37, 38: Create a new Employee object (which is a CustomDatum object) and define an employee name and employee number for it.

Lines 40-47: Prepare a statement to insert the new Employee object into the database. The setObject() method indicates that the object will be inserted into the first index position and that the underlying type of the Employee object is oracle.sql.STRUCT.
Creating Signed Applets

This section presents an example of a signed applet which uses the JDBC Thin driver to connect to and query a database. The code used in the applet was created with Oracle JDeveloper and complies with JDK 1.1.2 and JDBC 1.22. Signed applets are also browser-specific; the applet defined in this section works with the Netscape 4.x browser.

The applet displays a user interface that lets you connect to a local or a remote database, depending on whether you press the "Local" or "Remote" button. The applet queries the selected database for the contents of a specified row and displays the results.

If you want to try this example on your own system, you must provide this information:

- Obtain a copy of the Capabilities classes from Netscape and an object-signing certificate. You can find instructions for this at:
  

  Follow the instructions for obtaining a certificate and downloading the classes. The example in this section requires the Capabilities classes Principle.class, Privilege.class, PrivilegeManager.class, and PrivilegeTable.class.

In the applet code, replace the following strings:

- Replace <local database connect string> with the connect string for the local database. For example:
  
  "jdbc:oracle:thin:@myServer.us.oracle.com:1521:orcl", "scott","tiger"

- Replace <select on row of local table> with a SQL SELECT statement on a row in a table in the local database. For example:
  
  SELECT * FROM EMP WHERE ENAME = 'Mary'

Lines 49-54: Select the contents of the employee_table. Cast the results to an OracleResultSet so that the getCustomDatum() method can be used on it.

Lines 54-58: Iterate through the result set, getting the contents of the Employee objects and printing the employee names and employee numbers.

Lines 58-62: Close the result set, statement, and connection objects.
Replace `<remote database connect string>` with the connect string for the remote database. For example:
"jdbc:oracle:thin:@yourServer.us.oracle.com:1521:orcl", "scott","tiger"

Replace `<select on row of remote table>` with a SQL SELECT statement on a table in the remote database. For example:

```
SELECT * FROM EMP WHERE ENAME = 'Bob'
```

This applet uses only the Java AWT components and JDBC.
public MainApplet() {
}

// Initialize the applet
public void init() {
    try { jbInit(); } catch (Exception e) { e.printStackTrace(); }
    try {
        PrivilegeManager.enablePrivilege("UniversalConnect");
        PrivilegeManager.enablePrivilege("UniversalListen");
        PrivilegeManager.enablePrivilege("UniversalAccept");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// Component initialization
public void jbInit() throws Exception{
    this.setBounds(new Rectangle(0, 0, 400, 400));
    panel1.setBackground(Color.lightGray);
    panel1.setLayout(borderLayout3);
    this.setSize(new Dimension(372, 373));
    labelTitle.setBackground(Color.lightGray);
    labelTitle.setFont(new Font("Dialog", 0, 12));
    labelTitle.setAlignment(1);
    labelTitle.setText("Oracle Thin JDBC Driver Sample Applet");
    button1.setLabel("Local");
    panel3.setBackground(Color.lightGray);
    statusBar1.setBackground(Color.lightGray);
    statusBar1.setText("Ready");
    button2.setLabel("Remote");
    button2.addActionListener(new MainApplet_button2_actionAdapter(this));
    panel3.setLayout(borderLayout4);
    button1.addActionListener(new MainApplet_button1_actionAdapter(this));
    panel2.setLayout(borderLayout2);
    this.setLayout(borderLayout1);
    this.add(panel1, BorderLayout.NORTH);
    panel1.add(button1, BorderLayout.WEST);
    panel1.add(labelTitle, BorderLayout.CENTER);
    panel1.add(button2, BorderLayout.EAST);
    this.add(panel2, BorderLayout.CENTER);
    panel2.add(txtArResults, BorderLayout.CENTER);
    this.add(panel3, BorderLayout.SOUTH);
    panel3.add(statusBar1, BorderLayout.NORTH);
}
//Start the applet
public void start() {
}

//Stop the applet
public void stop() {
}

//Destroy the applet
public void destroy() {
}

//Get Applet information
public String getAppletInfo() {
    return "Applet Information";
}

//Get parameter info
public String[][] getParameterInfo() {
    return null;
}

//Main method
static public void main(String[] args) {
    MainApplet applet = new MainApplet();
    applet.isStandalone = true;
    Frame frame = new Frame();
    frame.setTitle("Applet Frame");
    frame.add(applet, BorderLayout.CENTER);
    applet.init();
    applet.start();
    frame.pack();
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    frame.setLocation((d.width - frame.getSize().width) / 2, (d.height - frame.getSize().height) / 2);
    frame.setVisible(true);
}

void button1_actionPerformed(ActionEvent e) {
    // Handler for "Local" Button.
    // Here is where we connect to local database
    StringBuffer b = new StringBuffer();
try {
    DriverManager.registerDriver ( new oracle.jdbc.driver.OracleDriver ());
    b.append ("DriverManager.registerDriver\r\n");
} catch (SQLException oe) {
    statusBar1.setText("registerDriver: Caught SQLException");
} catch (ClassNotFoundException oe) {
    statusBar1.setText("registerDriver: Caught ClassNotFoundException");
}

int numRows = 0;
try {
    statusBar1.setText("Executing Query on Local Database ...");
    Connection conn = DriverManager.getConnection ("jdbc:oracle:thin:<local database connect string>");
    b.append ("[DriverManager.getConnection] \r\n");
    Statement stmt = conn.createStatement ();
    b.append ("[conn.createStatement] \r\n");
    ResultSet rset = stmt.executeQuery ("<select on row of local table>");
    b.append ("[stmt.executeQuery] \r\n");
    b.append("SQL> \r\n");
    b.append("DSCR\n--------------------------------------\r\n");
    while (rset.next ()) {
        String ename = rset.getString (1);
        b.append (ename);
        b.append ("\r\n");
        numRows++;
    } // [end while rset.next() loop]
    statusBar1.setText("Query Done.");
} catch (SQLException SQLE) {
    statusBar1.setText ("Caught SQLException!");
    SQLE.printStackTrace();
} finally {
    b.append("\r\n");
    b.append(String.valueOf(numRows) + " rows selected.\r\n");
    txtArResults.setText ( b.toString ());
}

// End JDBC Code

void button2_actionPerformed(ActionEvent e) {
// Handler for the "Remote" Button
//
StringBuffer b = new StringBuffer();

try {
    DriverManager.registerDriver ( new oracle.jdbc.driver.OracleDriver ());
    b.append("DriverManager.registerDriver\n");
    } catch (SQLException oe) {
    statusBar1.setText("registerDriver: Caught SQLException");
    } catch (ClassNotFoundException oe) {
    statusBar1.setText("registerDriver: Caught ClassNotFoundException");
    }

int numRows = 0;  // line 183
try {
    statusBar1.setText("Executing Query on Remote Database ...");
    try {
        PrivilegeManager.enablePrivilege("UniversalConnect");
        b.append("enablePrivilege(UniversalConnect)\n");
        PrivilegeManager.enablePrivilege("UniversalListen");
        b.append("enablePrivilege(UniversalListen)\n");
        PrivilegeManager.enablePrivilege("UniversalAccept");
        b.append("enablePrivilege(UniversalAccept)\n");
        Connection conn = DriverManager.getConnection ("jdbc:oracle:thin:<remote database connect string>");
        b.append("DriverManager.getConnection\n");
        Statement stmt = conn.createStatement ();
        b.append("conn.createStatement\n");
        ResultSet rset = stmt.executeQuery ("<select on row of remote table>");
        b.append("stmt.executeQuery\n");
        b.append("SQL> <select on row of remote table>\n\n");
        while (rset.next () ) {
            String ename = rset.getString (1);
            b.append (ename);
            b.append ("\n");
            numRows++;
        }  // [end while rset.next() loop]
    statusBar1.setText("Query Done.");
}
Lines 6-11: Import the needed files.
**JDBC versus SQLJ Sample Code**

This section contains a side-by-side comparison of two versions of the same sample code: 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()` which SELECTs into a table and returns an employee’s address based on the employee’s number, and `updateAddress()` which 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 schema in the database and populate the tables.
- 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.

Both versions of the sample code reference objects and tables created by the `ObjectDemo.sql` script.

---

**Note:** The JDBC and SQLJ versions of the sample code are only code snippets. 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.

```sql
/*** 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
  (   
    emprnumber    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 (emprnumber, 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.

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

/**
 * This is what we have to do in JDBC
 **/
public class SimpleDemoJDBC
{

//TO DO: make a main that calls this

public Address getEmployeeAddress(int empno, Connection conn)
  throws SQLException
{
  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 7-39, 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
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 type code 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 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.
JDBC versus SQLJ Sample Code

- 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.

```java
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;
        }
```
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.

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.)

The `getEmployeeAddress()` method returns the `addr` object.

The `updateAddress()` method also uses the default connection context instance.

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`.

The `updateAddress()` method returns the `addr` object.

**Coding Requirements of the SQLJ Version**

Note the following coding requirements for the SQLJ version of the sample code:

- An explicit connection is not required; SQLJ assumes that a default connection context has 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 (for closing statements or result sets, for example) is required.
■ 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 out-parameters.
■ SQLJ syntax is simpler; for example, SELECT...INTO is supported and OBDC-style escapes are not used.
This chapter contains detailed JDBC reference information, including the following topics:

- Valid SQL-JDBC Datatype Mappings
- Supported SQL and PL/SQL Datatypes
- NLS Character Set Support
- Related Information
Valid SQL-JDBC Datatype Mappings

Table 3–1 and Table 3–2 in Chapter 3 describe the default mappings between Java classes and SQL datatypes that are supported by the Oracle JDBC drivers. Compare the contents of the **Standard JDBC Datatypes**, **Java Native Datatypes** and **Oracle SQL Datatypes** columns in Table 3–1 and Table 3–2 with the contents of Table 8–1 below.

Table 8–1 lists all of the possible Java classes 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 as an `oracle.sql.CHAR`, use `getCHAR()`. To materialize it as a `java.math.BigDecimal`, use `getBigDecimal()`.

**Table 8–1 Valid SQL Datatype-Java Class Mappings**

<table>
<thead>
<tr>
<th>This SQL datatype:</th>
<th>Can be materialized as these Java classes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR, NCHAR, VARCHAR2, NVARCHAR2, LONG</td>
<td><code>oracle.sql.CHAR</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.String</code></td>
</tr>
<tr>
<td></td>
<td><code>java.sql.Date</code></td>
</tr>
<tr>
<td></td>
<td><code>java.sql.Time</code></td>
</tr>
<tr>
<td></td>
<td><code>java.sql.Timestamp</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Byte</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Short</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Integer</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Long</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Float</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Double</code></td>
</tr>
<tr>
<td></td>
<td><code>java.math.BigDecimal</code></td>
</tr>
<tr>
<td></td>
<td><code>byte, short, int, long, float, double</code></td>
</tr>
<tr>
<td>DATE</td>
<td><code>oracle.sql.DATE</code></td>
</tr>
<tr>
<td></td>
<td><code>java.sql.Date</code></td>
</tr>
<tr>
<td></td>
<td><code>java.sql.Time</code></td>
</tr>
<tr>
<td></td>
<td><code>java.sql.Timestamp</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.String</code></td>
</tr>
<tr>
<td>This SQL datatype:</td>
<td>Can be materialized as these Java classes:</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>NUMBER</td>
<td>oracle.sql.NUMBER</td>
</tr>
<tr>
<td></td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td></td>
<td>java.lang.Short</td>
</tr>
<tr>
<td></td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td></td>
<td>java.lang.Long</td>
</tr>
<tr>
<td></td>
<td>java.lang.Float</td>
</tr>
<tr>
<td></td>
<td>java.lang.Double</td>
</tr>
<tr>
<td></td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td></td>
<td>byte, short, int, long, float, double</td>
</tr>
<tr>
<td>RAW, LONG RAW</td>
<td>oracle.sql.RAW</td>
</tr>
<tr>
<td></td>
<td>byte[]</td>
</tr>
<tr>
<td>ROWID</td>
<td>oracle.sql.CHAR</td>
</tr>
<tr>
<td></td>
<td>oracle.sql.ROWID</td>
</tr>
<tr>
<td></td>
<td>java.lang.String</td>
</tr>
<tr>
<td>BFILE</td>
<td>oracle.sql.BFILE</td>
</tr>
<tr>
<td>BLOB</td>
<td>oracle.sql.BLOB</td>
</tr>
<tr>
<td></td>
<td>oracle.jdbc2.Blob</td>
</tr>
<tr>
<td>CLOB, NCLOB</td>
<td>oracle.sql.CLOB</td>
</tr>
<tr>
<td></td>
<td>oracle.jdbc2.Clob</td>
</tr>
<tr>
<td>OBJECT</td>
<td>oracle.sql.STRUCT</td>
</tr>
<tr>
<td></td>
<td>oracle.sql.SqljData</td>
</tr>
<tr>
<td></td>
<td>oracle.jdbc2.Struct</td>
</tr>
<tr>
<td>REF</td>
<td>oracle.sql.REF</td>
</tr>
<tr>
<td></td>
<td>oracle.jdbc2.Ref</td>
</tr>
<tr>
<td>TABLE (nested), VARRAY</td>
<td>oracle.sql.ARRAY</td>
</tr>
<tr>
<td></td>
<td>oracle.jdbc2.Array</td>
</tr>
<tr>
<td>any of the above SQL types</td>
<td>oracle.sql.CustomDatum or oracle.sql.Datum</td>
</tr>
</tbody>
</table>
Notes:

- The type UROWID is not supported.
- `oracle.sql.Datum` is an abstract class. The value passed to a parameter of type `oracle.sql.Datum` must be of the Java type corresponding to the SQL type. Likewise, the value returned by a method with return type `oracle.sql.Datum` must be of the Java type corresponding to the 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 8–2 describes Oracle JDBC driver and SQLJ support for SQL datatypes.

<table>
<thead>
<tr>
<th>SQL Datatype</th>
<th>Supported by JDBC Drivers?</th>
<th>Supported by SQLJ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFILE</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>BLOB</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>CHAR</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>CLOB</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DATE</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>NCHAR</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>NCHAR VARYING</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>NUMBER</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>NVARCHAR2</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>RAW</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>REF</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>ROWID</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>UROWID</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
Table 8–3 describes Oracle JDBC driver and SQLJ support for the ANSI-supported SQL datatypes.

Table 8–3  Support for ANSI-Supported SQL Datatypes

<table>
<thead>
<tr>
<th>ANSI-Supported SQL Datatypes</th>
<th>Supported by JDBC Drivers?</th>
<th>Supported by SQLJ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DEC</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>FLOAT</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>INT</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>INTEGER</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>NATIONAL CHARACTER</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>NATIONAL CHARACTER VARYING</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>NATIONAL CHAR</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>NATIONAL CHAR VARYING</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>NCHAR</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>NCHAR VARYING</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>REAL</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 8–4 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 8–4 Support for PL/SQL Datatypes

<table>
<thead>
<tr>
<th>PL/SQL Datatypes</th>
<th>Supported by JDBC Drivers?</th>
<th>Supported by SQLJ?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalar Types:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary integer</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>dec</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>decimal</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>double precision</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>float</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>int</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>integer</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>natural</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>naturaln</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>number</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>numeric</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>pls_integer</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>positive</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>positiven</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>real</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>signtype</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>smallint</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Scalar Character Types:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>character</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>long</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>long raw</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>nchar</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>nvarchar2</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>raw</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>rowid</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
### Table 8–4  Support for PL/SQL Datatypes (Cont.)

<table>
<thead>
<tr>
<th>PL/SQL Datatypes</th>
<th>Supported by JDBC Drivers?</th>
<th>Supported by SQLJ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>urowid</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>varchar</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>varchar2</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>boolean</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>date</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Composite Types:**

- record: no, no
- table: no, no
- varray: yes, yes

**Reference Types:**

- REF CURSOR: yes, yes
- REF object type: yes, yes

**LOB Types:**

- BFILE: yes, yes
- BLOB: yes, yes
- CLOB: yes, yes
- NCLOB: no, no

**Notes:**

- The types natural, natural n, positive, positive n, and signtype are subtypes of binary integer.
- The types dec, decimal, double precision, float, int, integer, numeric, real, and smallint, are subtypes of number.
NLS Character Set Support

On the client, the Oracle JDBC OCI and Thin drivers support all Oracle NLS character sets. On the server, the Oracle JDBC Server driver supports only two Oracle NLS character sets: US7ASCII (ASCII 7-bit American) and WE8ISO8859P1 (ISO 8859-1 West European or "ISO-Latin 1").

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 and SQLJ, Java technology, the Java Developer’s Kit APIs (for versions 1.2 and 2.0), and resources to help you write applets.

Oracle JDBC Drivers and SQLJ

Oracle JDBC Driver Home Page (Oracle Corporation)
http://www.oracle.com/st/products/jdbc/

Oracle JDBC Driver Download Page (Oracle Corporation)
http://www.oracle.com/products/free_software/index.html#jdbc8

Oracle SQLJ Home Page (Oracle Corporation)

Java Technology

Java Technology Home Page (Sun Microsystems, Inc.):
http://www.javasoft.com/

Java Development Kit 1.1 (JDK1.1) (Sun Microsystems, Inc.):
http://java.sun.com/products/jdk/1.1/

Java Platform JDK1.1 Core API Specification (Sun Microsystems, Inc.):
http://www.javasoft.com/products/jdk/1.1/docs/api/packages.html

Java Development Kit 1.2 (JDK1.2) (Sun Microsystems, Inc.):
Related Information

Java Platform JDK1.2 Core API Specification (Sun Microsystems, Inc.):
http://www.javasoft.com/products/jdk/1.2/docs/api/index.html

Signed Applets

Introduction to Capabilities Classes (Netscape Communications Corp.):

Object-Signing Resources (Netscape Communications Corp.):

Signed Applet Example (Sun Microsystems, Inc.):
This appendix lists the error messages that the Oracle JDBC drivers can return. "Cause" and "Action" information for each message will be provided in a later release.

byte array not long enough

can only describe a query

cannot do new defines until the current result set is closed

cannot set row prefetch to zero

char array not long enough

character set not supported

closed connection

closed LOB

closed resultset

closed statement
cursor already initialized
error in defines.isNull ()

error in type descriptor parse

exception in OracleNumber

exhausted resultset

fail to construct descriptor

fail to convert between UTF8 and UCS2

fail to convert to internal representation

inconsistent Java and SQL object types

Internal error: attempt to access bind values beyond the batch value

Internal error: data array not allocated

Internal error: invalid index for data access

Internal error: invalid NLS Conversion ratio

invalid batch value

invalid character encountered in

invalid column name

invalid column type

invalid cursor

invalid dynamic column
invalid row prefetch

invalid stream maximum size

malformed SQL92 string at position

missing defines

missing defines at index

missing descriptor

missing IN or OUT parameter at index:

no data read

no such element in vector

non supported character set

non-supported SQL92 token at position

numeric overflow

only one RPA message is expected

only one RXH message is expected

parameter type conflict

protocol violation

received more RXDs than expected
REF cursor is invalid

resultSet.next was not called

setAutoClose: only support auto close mode on

setReadOnly: read-only connections not supported

setTransactionIsolation: only supports TRANSACTION_READ_UNCOMMITTED

statement timed out

statement was cancelled

stream has already been closed

sub protocol must be specified in connection URL

the LOB locator is not valid

the size is not valid

This API cannot be used for non-UDT types

this ref is not valid

undefined type

unsupported column type

unsupported feature
### A

- addBatch() method, restrictions on, 4-100
- ANO (Advanced Networking Option), 3-27
- APPLET HTML tag, 5-19
- **applets**
  - coding, 5-7
    - for JDK 1.0.2 browser, 5-8
    - for JDK 1.1.1 browser, 5-8
  - connecting to a database, 5-9
  - deploying in an HTML page, 5-19
  - packages needed, 5-7
  - packaging, 5-17
    - for JDK 1.0.2 browser, 5-17
    - for JDK 1.1.1 browser, 5-18
  - packaging and deploying, 3-28
- **signed applets**
  - browser security, 5-20
  - example program, 7-31
  - object-signing certificate, 5-14
  - using, 5-13
  - using with firewalls, 5-14
  - working with, 5-7
- ARCHIVE, parameter for APPLET tag, 5-20
- **ARRAY**
  - class, 4-14
  - descriptors, 4-15
  - objects, creating, 4-15, 4-16
- array descriptor
  - creating, 4-93
- ArrayDescriptor object, 4-15, 4-93
  - creating, 4-16
  - get methods, 4-17
- arrays defined, 4-87
  - example program, 7-16
  - getting, 4-92
  - named, 4-87
  - passing to callable statement, 4-94
  - retrieving from a result set, 4-88
  - retrieving partial arrays, 4-91
  - using type maps, 4-94
  - working with, 4-87
- AUTHENTICATION_LEVEL parameter, 5-12
  - auto-commit mode
    - disabling, 6-5
    - result set behavior, 6-5

### B

- batching values, 4-97, 6-6
  - and streaming data, 4-100
  - connection-wide, 4-103
  - default batch size, 4-100
  - not supported by OracleCallableStatement, 4-100
  - overriding default value, 4-102
  - restrictions on, 4-100
  - setting batch value, 4-101
- BFILE
  - accessing data, 4-60
    - class, 4-17
    - creating and populating columns, 4-58
    - defined, 3-24
    - example program, 7-10
    - locators, 4-55
    - getting from a result set, 4-55
    - getting from callable statement, 4-56
passing to callable statements, 4-56
passing to prepared statements, 4-56
manipulating data, 4-60
reading data, 4-57
working with, 4-45
BFILE locator, selecting, 4-18
BLOB, 4-48
class, 4-17
creating and populating, 4-52
creating columns, 4-52
getting locators, 4-46
locators
getting from result set, 4-46
selecting, 4-18
manipulating data, 4-54
populating columns, 4-53
reading data, 4-48, 4-50
working with, 4-45
writing data, 4-51
Boolean parameters, restrictions, 6-7
browser security, 5-20

callable statement
getting a BFILE locator, 4-56
getting LOB locators, 4-47
passing BFILE locator, 4-56
passing LOB locators, 4-48
using getOracleObject() method, 4-36
casting return values, 4-39
catalog arguments (DatabaseMetaData), 4-116
CHAR
object, creating, 4-20
CHAR class, 4-19
conversions with KPRB driver, 5-25
CHAR columns
space padding, 6-7
character sets, 4-21
conversions with KPRB driver, 5-25
Class.forName method, 3-3
CLASSPATH, specifying, 2-7
clearDefines() method, 4-105
client installation, 3-27
CLOB
class, 4-17
creating and populating, 4-52
creating columns, 4-52
locators, 4-46
getting from result set, 4-46
passing to callable statements, 4-48
passing to prepared statement, 4-48
locators, selecting, 4-18
manipulating data, 4-54
populating columns, 4-53
reading data, 4-48, 4-50
working with, 4-45
writing data, 4-51
close() method, 4-25, 4-26, 4-27, 6-7
for database connection, 5-23
closeFile() method, 4-19
CMAN.ORA file, creating, 5-11
CODE, parameter for APPLET tag, 5-19
CODEBASE, parameter for APPLET tag, 5-19
collections, 4-87
collections (nested tables and arrays), 4-15
column types
redefining, 4-97, 4-105
restrictions on, 4-105
COMMIT operation, 5-24
connect string
for KPRB driver, 5-23
for the Oracle8 Connection Manager, 5-12
connection
closing, 3-10
from KPRB driver, 3-26
opening, 3-3
opening for JDBC OCI driver, 3-6
opening for JDBC Thin driver, 3-7
Properties object, 3-6
Connection Manager, 3-28, 5-9, 5-10
browser security, 5-20
installing, 5-11
starting, 5-12
using multiple managers, 5-13
writing the connect string, 5-12
connection properties, 4-109
database, 4-110
defaultBatchValue, 4-110
defaultRowPrefetch, 4-110
password, 4-110
put() method, 4-110
remarksReporting, 4-110
user, 4-110
connections
read-only, 4-116
constants for SQL types, 4-28
CREATE DIRECTORY statement
for BFILEs, 4-58
CREATE TABLE statement
to create BFILE columns, 4-58
to create BLOB, CLOB columns, 4-52
create() method
for CustomDatumFactory interface, 4-75
createDescriptor() method, 4-13
createStatement() method, 4-24
CursorName
limitations, 4-115
cursors, 6-7
custom Java classes
defining, 4-65
custom Java types
creating, 7-20, 7-26
CustomDatum interface, 4-3
advantages, 4-65
eample program, 7-26
reading data, 4-79
writing data, 4-80

data conversions, 4-32
LONG, 3-15
LONG RAW, 3-15
data streaming
avoiding, 3-18
eample program, 7-2
database
connecting
from an applet, 5-10
through multiple Connection Managers, 5-13
with KPRB, 5-22
connection testing, 2-8
database connection
connection property, 4-110
database URL
including userid and password, 3-5
database URL, specifying, 3-4
DatabaseMetaData calls, 4-116
DatabaseMetaData class, 5-28	entry points for applets, 5-18
datatype classes, 4-7
datatype mappings, 3-11
datatypes
Java, 3-11, 3-12
Java native, 3-11, 3-12
JDBC, 3-11, 3-12
JDBC extensions for Oracle SQL datatypes, 3-12
Oracle SQL, 3-11, 3-12
DATE class, 4-22
DBMS_LOB package, 4-49
debugging JDBC programs, 6-9
DEFAULT_CHARSET character set value, 4-21
defaultBatchValue connection property, 4-110
defaultConnection() method, 5-22
defaultRowPrefetch connection property, 4-110
defineColumnType() method, 3-19, 4-25, 4-106
dnldthin sub-protocol, 5-8
DriverManager class, 3-3
dynamic SQL, 1-2

E
eecryption
applets, 3-28
applications, 3-27
evironment variables
specifying, 2-7
error handling and messages, 3-25
eexception trapping, 6-9
eexecuteBatch() method
restrictions on, 4-100
executeQuery() method, 4-25
eecuteUpdate() method, 4-101
extensions to JDBC, Oracle, 4-1
external file
defined, 3-24
F
finalizer methods, 6-7
firewalls
  configuring for applets, 5-15
  connect string, 5-16
  described, 5-15
  required rule list items, 5-15
  using with applets, 3-28, 5-14
floating-point compliance, 4-115
function call syntax, SQL92 syntax, 5-30

G
getARRAY() method, 4-88
getArray() method, 4-15, 4-88
  using type maps, 4-90
getArrayDescriptor() method, 4-15
getAsciiOutputStream() method, 4-18
  for writing CLOB data, 4-50
getAsciiStream() method, 4-18
  for reading CLOB data, 4-49
getAttributes() method, 4-11, 4-63
  used by Structs, 4-69
getBaseName() method, 4-17
getBaseType() method, 4-15, 4-17, 4-92
getBaseTypeName() method, 4-14, 4-15
  used with object references, 4-83
getBinaryOutputStream() method, 4-18
  for writing BLOB data, 4-49
getBinaryStream() method, 3-17, 4-18, 4-19
  for reading BFILE data, 4-57
  for reading BLOB data, 4-49
getBLOB() method, 4-46
getBytes() method, 3-18, 4-10, 4-18, 4-19
getCharacterOutputStream() method, 4-18
  for writing CLOB data, 4-50
getCharacterStream() method, 4-18
  for reading CLOB data, 4-49
getChars() method, 4-18
getChunkSize() method, 4-53
getCLOB() method, 4-46
getColumnCount() method, 4-28
getColumnType() method, 4-28, 4-44
getColumnTypeName() method, 4-28, 4-44
getConnection() method, 4-28, 4-44
  connection properties, 4-109
  for applets, 5-8
getCursor() method, 4-113, 4-114
getCustomDatum() method, 4-76, 4-79
getDefaultExecuteBatch() method, 4-24
getDefaultRowPrefetch() method, 4-24, 4-98
getDescriptor() method, 4-12
getDirAlias() method, 4-19, 4-60
getPreparedStatement() method, 4-26, 4-101
  returning current batch value, 4-103
getMap() method, 4-12
getName() method, 4-19, 4-60
getNumericFunctions() method, 5-28
getObject() method
  casting return values, 4-39
  for BLOBs and CLOBs, 4-46
  for CustomDatum objects, 4-76
  for object references, 4-84
  for SQLInput streams, 4-70
  for SQLOutput streams, 4-71
  for Struct objects, 4-64
  return types, 4-34, 4-36
  to get BFILE locators, 4-55
  to get Oracle objects, 4-63
  used with CustomDatum interface, 4-80
getOracleArray() method, 4-15, 4-88, 4-92
getOracleAttributes() method, 4-12, 4-64
getOracleObject() method, 4-26, 4-27
  casting return values, 4-39
  for BLOBs and CLOBs, 4-46
  return types, 4-35, 4-36
  using in callable statement, 4-36
  using in result set, 4-35
getProcedureColumns() method, 4-108
getProcedures() method, 4-108
getREF() method, 4-85
getRemarksReporting() method, 4-25
getResultSet() method, 4-15, 4-25
getRowPrefetch() method, 4-25, 4-98
getSQLTypeName() method, 4-12, 4-15, 4-63, 4-92
getString() method, 4-21
  to get ROWIDs, 4-111
getStringFunctions() method, 5-28
getStringWithReplacement() method, 4-21
getSTRUCT() method, 4-64
getSubString() method, 4-19
  for reading CLOB data, 4-50
getsystemFunctions() method, 5-28
getTableName() method, 4-28
g getTimeDateFunctions() method, 5-28
getTransactionIsolation() method, 4-24, 6-13
gettypeMap() method, 4-25, 4-67
getValue() method, 4-14
  for object references, 4-84
genXXX() methods
  casting return values, 4-39
  for specific datatypes, 4-37

H
HEIGHT, parameter for APPLET tag, 5-19
HTML tags, to deploy applets, 5-19
HTTP protocol, 1-5

I
IEEE 754 floating-point compliance, 4-115
INSERT INTO statement
  for creating BFILE columns, 4-59
inserts to database, accumulating, 6-6
installation
  client, 3-27
directories and files, 2-6
  verifying on the client, 2-6
instanceOf() method, 4-35
initWith() method, 4-10
isConvertibleTo() method, 4-12

J
Java
  compiling and running, 2-7
datatypes, 3-11, 3-12
native datatypes, 3-11, 3-12
stored procedures, 3-25
stream data, 3-14
Java Sockets, 2-2
java.math, Java math packages, 3-2
java.sql, JDBC packages, 3-2
java.sql.SQLException() method, 3-25
java.sql.Types class, 4-106
java.util.Dictionary class
  used by type maps, 4-67
java.util.Hashtable class
  used by type maps, 4-67
java.util.Map class, 4-91
JDBC
  and IDEs, 1-7
  and Oracle Application Server, 1-6
basic program, 3-2
datatypes, 3-11, 3-12
defined, 1-2
error handling and messages, 3-25
guidelines for using, 1-3
importing packages, 3-2
limitations of Oracle extensions, 4-115
Oracle extensions, 1-6
sample files, 2-7
testing, 2-8
versions supported, 1-6
JDBC calls, logging, 6-10
JDBC drivers
  and NLS, 5-2
applets, 3-27
applications, 3-27
basic architecture, 1-4
choosing a driver for your needs, 2-4
common features, 2-2
common problems, 6-6
compatibilities, 2-5
determining driver version, 2-8
registering, 3-3
registering for an applet, 5-7
requirements, 2-5
restrictions, 6-7
SQL92 syntax, 5-26
JDBC KPRB driver
  architecture, 1-6
described, 2-4
JDBC mapping (for attributes), 4-82
JDBC OCI driver
  applications, 3-27
  architecture, 1-5
  described, 2-3
  NLS considerations, 5-3

JDBC Thin driver
  applets, 3-27, 5-7
  applications, 3-27
  architecture, 1-5
  described, 2-2
  NLS considerations, 5-4

JdbcCheckup program, 2-8
JDeveloper, 1-7

JDK
  version considerations, 5-20
  versions supported, 1-6

JPublisher utility, 4-3, 4-65
  data mapping options, 4-82
  using with JDBC, 4-82

K
  KPRB driver
    connection string for, 5-23
    connection to database, 5-22
    described, 5-22
    NLS considerations, 5-4
    relation to the SQL engine, 5-22
    session context, 5-23
    support for NLS, 5-25
    testing, 5-24
    transaction context, 5-23

L
  LD_LIBRARY_PATH variable, specifying, 2-7
  length() method, 4-18, 4-19
  LIKE escape characters, SQL92 syntax, 5-29
  limitations, 4-116
  LOB
    defined, 3-23
    locators, 4-45
    reading data, 4-48
    working with, 4-45
  LOB locators
    getting from callable statements, 4-47
    passing, 4-47
  LOBs
    example program, 7-4
    locators
      getting for BFILEs, 4-55
      getting for BLOBs, 4-46
      getting for CLOBs, 4-46
      LOB, 4-45
      passing to callable statements, 4-48
      passing to prepared statement, 4-48
  LONG
    data conversions, 3-15
  LONG RAW
    data conversions, 3-15

M
  makeDatumArray() method, 4-12
  memory leaks, 6-7
  multi-threaded applications
    on the client, 6-2

N
  named arrays, 4-87
    defined, 4-15
  National Language Support (NLS), 4-21
  Net8
    name-value pair, 3-4
    protocol, 1-5
    network events, trapping, 6-10
    next() method, 4-27
  NLS
    and JDBC drivers, 5-2
    conversions, 5-2
      data size restrictions, 5-5
      for JDBC OCI drivers, 5-3
      for JDBC Thin drivers, 5-4
      for KPRB driver, 5-4
      Java methods that employ, 5-2
      using, 5-2
  NLS Ratio, 5-5
  NLS_LANG environment variable, 5-3
  NULL data
converting, 4-33
NUMBER class, 4-22

Object JDBC mapping (for attributes), 4-82
object references
- accessing object values, 4-84, 4-85
- defined, 4-83
- passing to callable statement, 4-85
- passing to prepared statements, 4-86
- redefining columns containing, 4-105
- updating object values, 4-84, 4-85
- working with, 4-83
openFile() method, 4-19
optimization, performance, 6-5
Oracle Application Server, 1-6
Oracle datatypes
- using, 4-32
Oracle extensions
- datatype support, 4-2
- limitations, 4-115
  - catalog arguments to DatabaseMetaData calls, 4-116
  - CursorName, 4-115
  - IEEE 754 floating-point compliance, 4-115
  - PL/SQL TABLE, BOOLEAN, RECORD types, 4-115
  - read-only connection, 4-116
  - SQL92 outer join escapes, 4-115
  - SQLWarning class, 4-116
- object support, 4-3
- packages, 4-2
- performance extensions, 4-97
- result sets, 4-33
- schema naming support, 4-4
- statements, 4-33
- to JDBC, 4-1
Oracle mapping (for attributes), 4-82
Oracle objects
- and JDBC, 4-62
  - converting with CustomDatum interface, 4-75
  - converting with SQLData interface, 4-69
  - defining with Java classes, 4-65
  - getting with getObject() method, 4-63
Java classes which support, 4-62
reading data by using SQLData interface, 4-72
working with, 4-62
writing data by using SQLData interface, 4-74
Oracle SQL datatypes, 3-11, 3-12
Oracle8 Connection Manager, 5-9
OracleCallableStatement class, 4-26
getXXX() methods, 4-37
registerOutParameter() method, 4-42
OracleConnection class, 4-24
OracleDatabaseMetaData class, 5-28
and applets, 5-18
OracleDriver class, 4-24
oracle.jdbc2 package, described, 4-6
oracle.jdbc2.Struct class, 4-10, 4-63
getAttribute() method, 4-63
getSQLTypeName() method, 4-63
oracle.jdbc.driver package, 4-22
stream classes, 4-28
oracle.jdbc.driver, Oracle JDBC extensions, 3-3
oracle.jdbc.driver.OracleCallableStatement class, 4-26
close() method, 4-27
getOracleObject() method, 4-26
getXXX() methods, 4-26
registerOutParameter() method, 4-27
setNull() method, 4-27
setOracleObject() methods, 4-27
setXXX() methods, 4-27
oracle.jdbc.driver.OracleConnection class, 4-24
createStatement() method, 4-24
defaultExecuteBatch() method, 4-24
defaultRowFetch() method, 4-24
getRemarksReporting() method, 4-25
getTransactionIsolation() method, 4-24, 6-13
getTypeMap() method, 4-25
prepareCall() method, 4-24
prepareStatement() method, 4-24
setDefaultExecuteBatch() method, 4-24
setDefaultRowPrefetch() method, 4-25
setRemarksReporting() method, 4-25
setTransactionIsolation() method, 4-24, 6-13
setTypeMap() method, 4-25
oracle.jdbc.driver.OracleDriver class, 4-24, 5-7
oracle.jdbc.driver.OraclePreparedStatement
Index-8

class, 4-25
close() method, 4-26
getExecuteBatch() method, 4-26
setCustomDatum() method, 4-26
setExecuteBatch() method, 4-26
setNull() method, 4-26
setOracleObject() method, 4-26
setXXX() methods, 4-26
oracle.jdbc.driver.OracleResultSet class, 4-27
getOracleObject() method, 4-27
getXXX() methods, 4-27
next() method, 4-27
oracle.jdbc.driver.OracleResultSetMetaData class, 4-28, 4-44
columnCount() method, 4-28
getColumnName() method, 4-28
columnName() method, 4-28
columnType() method, 4-28
columnTypeName() method, 4-28
tableName() method, 4-28
using, 4-44
oracle.jdbc.driver.OracleStatement class, 4-25
close() method, 4-25
defineColumnType(), 4-25
executeQuery() method, 4-25
getRowPrefetch() method, 4-25
setPosition() method, 4-25
oracle.jdbc.driver.OracleTypes class, 4-28, 4-106
OraclePreparedStatement class, 4-25
OracleResultSet class, 4-27
getXXX() methods, 4-37
OracleResultSet object, 3-9
OracleResultSetMetaData class, 4-28
OracleServerDriver class
defaultConnection() method, 5-22
oracle.sql datatype classes, 4-7
oracle.sql package
data conversions, 4-32
described, 4-7
oracle.sql.ARRAY class, 4-87
and nested tables, 4-14
and VARRAYs, 4-14
getArray() method, 4-15
getArrayDescriptor() method, 4-15
getBaseType() method, 4-15
getBaseTypeName() method, 4-15
getOracleArray() method, 4-15
getResultSet() method, 4-15
getSQLTypeName() method, 4-15
oracle.sql.ArrayDescriptor class
getName() method, 4-17
getBaseType() method, 4-17
getBaseTypeName() method, 4-17
oracle.sql.BFILE class, 4-17
closeFile() method, 4-19
getBinaryStream() method, 4-19
getBytes() method, 4-19
getDirAlias() method, 4-19
getName() method, 4-19
length() method, 4-19
openFile() method, 4-19
position() method, 4-19
oracle.sql.BLOB class, 4-17
getBinaryOutputStream() method, 4-18
getBinaryStream() method, 4-18
getBytes() method, 4-18
length() method, 4-18
position() method, 4-18
putBytes() method, 4-18
oracle.sql.CHAR class, 4-19, 5-25
getString() method, 4-21
getStringWithReplacement() method, 4-21
toString() method, 4-21
oracle.sql.CharacterSet class, 4-20
oracle.sql.CLOB class, 4-17
getAsciiOutputStream() method, 4-18
getAsciiStream() method, 4-18
getCharacterOutputStream() method, 4-18
getCharacterStream() method, 4-18
getChars() method, 4-18
getSubString() method, 4-19
length() method, 4-19
position() method, 4-19
putChars() method, 4-19
putString() method, 4-19
supported character sets, 4-18
oracle.sql.CustomDatum interface, 4-75
oracle.sql.CustomDatumFactory interface, 4-75
oracle.sql.datatypes support, 4-10
oracle.sql.DATE class, 4-22
oracle.sql.Datum class, described, 4-7
oracle.sql.NUMBER class, 4-22
OracleSql.parse() method, 5-30
oracle.sql.RAW class, 4-22
oracle.sql.REF class, 4-14, 4-83
  getBaseTypeName() method, 4-14
  getValue() method, 4-14
  setValue() method, 4-14
oracle.sql.REFCURSOR class, 4-112
oracle.sql.ROWID class, 4-10, 4-22, 4-111
oracle.sql.STRUCT class, 4-10, 4-63
  getConnection() method, 4-12
  getDescriptor() method, 4-12
  getMap() method, 4-12
  getOracleAttributes() method, 4-12
  getSQLTypeName() method, 4-12
  isConvertibleTo() method, 4-12
  makeJdbcArray() method, 4-12
  methods, 4-11
  getAttributes() method, 4-11
  setDatumArray() method, 4-12
  stringValue() method, 4-12
toBytes() method, 4-12
toClass() method, 4-12
toJDBC() method, 4-12
toSTRUCT() method, 4-12
oracle.sql.StructDescriptor class, 4-13
  createDescriptor() method, 4-13
OracleStatement class, 4-25
OracleTypes class, 4-28
OracleTypes.ARRAY class, 4-28, 4-44
OracleTypes.BFILE class, 4-29
OracleTypes.BLOB class, 4-28
OracleTypes.CLOB class, 4-28
OracleTypes.CURSOR variable, 4-113
OracleTypes.REF class, 4-28
OracleTypes.ROWID class, 4-29
OracleTypes.STRUCT class, 4-28, 4-44
outer joins, SQL92 syntax, 5-29

P

password connection property, 4-110
password, specifying, 3-4
PATH variable, specifying, 2-7
performance extensions
  batching updates, 4-100
  connection properties, 4-109
  prefetching rows, 4-98
  redefining column types, 4-105
  TABLE_REMARKS reporting, 4-108
to JDBC, 4-97
performance optimization, 6-5
  batching values, 6-6
  prefetching rows, 6-6
PL/SQL
  restrictions, 6-7
  space padding, 6-7
  stored procedures, 3-24
PL/SQL stored procedures, 3-24
PL/SQL types
  limitations, 4-115
  position() method, 4-18, 4-19
  prefetching rows, 4-97, 4-98, 6-6
  suggested default, 4-98
prepareCall() method, 4-24
prepared statement
  passing BFILE locator, 4-56
  passing LOB locators, 4-48
  using setObject() method, 4-41
  using setOracleObject() method, 4-41
prepareStatement() method, 4-24
printStackTrace() method, 6-9
put() method
  for Properties object, 4-110
  for type maps, 4-67
putBytes() method, 4-18
putChars() method, 4-19
putString() method, 4-19

Q

query, executing, 3-8

R

RAW class, 4-22
readSQL() method, 4-69, 4-70
  implementing, 4-70
REF class, 4-14
REFCURSORs, 4-112
  example program, 7-14
  materialized as result set objects, 4-112
reference classes, and JPublisher, 4-83
registerDriver() method, 4-24
registering Oracle JDBC drivers, class for, 4-24
registerOutParameter() method, 4-27, 4-42
remarksReporting connection property, 4-110
remarksReporting flag, 4-97
result set
  auto-commit mode, 6-5
  getting BFILE locators, 4-55
  getting LOB locators, 4-46
  metadata, 4-28
  Oracle extensions, 4-33
  using getOracleObject() method, 4-35
result set object
  closing, 3-9
result set, processing, 3-9
ResultSet class, 3-8
return types
  for getXXX() methods, 4-38
  getBLOB() method, 4-47
  getOracleObject() method, 4-36
return values
  casting, 4-39
ROLLBACK operation, 5-24
row prefetching, 4-98
and data streams, 3-23
ROWID class, 4-22
  CursorName methods, 4-115
defined, 4-111

S
scalar functions, SQL92 syntax, 5-28
schema naming conventions, 4-4
security, for browsers, 5-20
SELECT statement
  to retrieve object references, 4-84
  to select LOB locator, 4-54
sendBatch() method, 4-102
session context, 3-26
  for KPRB driver, 5-23
  setAutoCommit() method, 6-5
  setBFILE() method, 4-56
  setBLOB() method, 4-47
  setCLOB() method, 4-48
  setCursorName() method, 4-111, 4-115
  setCustomDatum() method, 4-26, 4-77, 4-81
  setDatumArray() method, 4-12
  setDefaultExecuteBatch() method, 4-24
  setDefaultRowPrefetch() method, 4-25, 4-98
  setDescriptor() method, 4-12
  setEscapeProcessing() method, 5-26
  setExecuteBatch() method, 4-26
  setLogStream() method, for logging JDBC calls, 6-10
  setMaxFieldSize() method, 4-106, 6-7
  setNull() method, 4-26, 4-27, 4-42
  setObject() method, 4-40
  setObject() method
    for BFILES, 4-56
    for BLOBs and CLOBs, 4-47
    for CustomDatum objects, 4-77
    for object references, 4-86
    to write object data, 4-81
    using in prepared statements, 4-41
  setOracleObject() method, 4-26, 4-27, 4-40
    for BFILES, 4-56
    for BLOBs and CLOBs, 4-47
    for Struct objects, 4-64
    using in prepared statements, 4-41
  setREF() method, 4-86
  setRemarksReporting() method, 4-25, 4-108
  setRowPrefetch() method, 4-25, 4-98
  setString() method
    to bind ROWIDs, 4-111
  setTransactionIsolation() method, 4-24, 6-13
  setTypeMap() method, 4-25, 4-68
  setValue() method, 4-14
  setXXX() methods, for specific datatypes, 4-41
signed applets, 3-28
SQL
  data converting to Java datatypes, 4-32
  primitive types, 4-7
  structured types, 4-7
types, constants for, 4-28
SQL engine
relation to the KPRB driver, 5-22
SQL syntax (Oracle), 5-26
SQL92 syntax, 5-26

function call syntax, 5-30
LIKE escape characters, 5-29
outer joins, 5-29
scalar functions, 5-28
time and date literals, 5-26
translating to SQL example, 5-30

SQLData interface, 4-3
advantages, 4-66
described, 4-69
example program, 7-20
Oracle implementation, 4-6
reading data from Oracle objects, 4-72
using with type map, 4-69
writing data from Oracle objects, 4-74

SQLException() method, 6-9

SQLInput interface, 4-69
described, 4-70
SQLInput streams, 4-70
SQL
advantages over JDBC, 1-3
guidelines for using, 1-3
SQLNET.ORA
parameters for tracing, 6-10
SQLOutput interface, 4-69
described, 4-70
SQLOutput streams, 4-71
SQLWarning class, limitations, 4-116
Statement object
closing, 3-28
creating, 3-8
statements
Oracle extensions, 4-33
static SQL, 1-2
stored procedures
Java, 3-25
PL/SQL, 3-24
stream classes, 4-28
stream data, 3-14, 4-48
CHAR columns, 3-19
closing, 3-23
example, 3-16
external files, 3-23
LOBs, 3-23
LONG columns, 3-14
LONG RAW columns, 3-14
multiple columns, 3-20
RAW columns, 3-19
row prefetching, 3-23
UPDATE/COMMIT statements, 4-50
VARCHAR columns, 3-19

stream data column
bypassing, 3-21

stringValue() method, 4-10, 4-12

STRUCT class, 4-10
STRUCT descriptor, 4-13
STRUCT object, 4-10
attributes, 4-11
casting, 4-63
creating, 4-13
embedded object, 4-14
nested objects, 4-11
using, 4-63

StructDescriptor object
creating, 4-13
get methods, 4-13
structured objects, 4-9
class for binding, 4-26

T

TABLE_REMARKS columns, 4-97
TABLE_REMARKS reporting
restrictions on, 4-108
TCP/IP protocol, 1-5, 3-7
time and date literals, SQL92 syntax, 5-26
TNSNAMES entries, 3-4
toBytes() method, 4-12
toClass() method, 4-12
toDatum() method
applied to CustomDatum objects, 4-65, 4-75
called by setCustomDatum() method, 4-81
toJDBC() method, 4-12
tojdbc() method, 4-10
toString() method, 4-21
toSTRUCT() method, 4-12
trace facility, 6-10
trace parameters
client-side, 6-11
server-side, 6-12
transaction context, 3-26
for KPRB driver, 5-23
TTC protocol, 1-5
type map, 4-3, 4-34, 4-65
  adding entries, 4-67
  and STRUCTs, 4-69
  creating a new map, 4-68
  defining mappings, 4-67
  described, 4-66
  used with arrays, 4-91
  used with SQLData interface, 4-69
  using with arrays, 4-94
type maps
  relationship to database connection, 5-23

U
updates to database, accumulating, 6-6
user connection property, 4-110
userid, specifying, 3-4
using, 5-10

V
VARCHAR2 columns, 6-7
varrays
  example program, 7-16

W
WIDTH, parameter for APPLET tag, 5-19
writeSQL() method, 4-69, 4-71
  implementing, 4-70