

---

## Basic Features

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

The following topics are discussed:

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

### First Steps in JDBC

This section describes how to get up and running with the Oracle JDBC drivers. When using the Oracle JDBC drivers, you must include certain driver-specific information in your programs. This section describes, in the form of a tutorial, where and how to add the information. The tutorial guides you through creating code to connect to and query a database from the client.

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

1. [Importing Packages](#)
2. [Opening a Connection to a Database](#)
3. [Creating a Statement Object](#)
4. [Executing a Query and Returning a Result Set Object](#)
5. [Processing the Result Set](#)
6. [Closing the Result Set and Statement Objects](#)
7. [Making Changes to the Database](#)
8. [Committing Changes](#)
9. [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, include the `import` statements shown in [Table 4–1](#) at the beginning of your program:

**Table 4–1 Import Statements for JDBC Driver**

Import statement	Required by
<code>import java.sql.*;</code>	standard JDBC packages
<code>import java.math.*;</code>	<code>BigDecimal</code> and <code>BigInteger</code> classes (you can omit this import if you don't use these classes)
<code>import oracle.jdbc.*;</code>	(optional) Oracle extensions to JDBC
<code>import oracle.jdbc.pool.*;</code>	
<code>import oracle.sql.*;</code>	

The Oracle packages listed as optional provide access to the extended functionality provided by the Oracle drivers, but are not required for the example presented in this section. For an overview of the Oracle extensions to the JDBC standard, see [Chapter 10, "Oracle Extensions"](#).

## Opening a Connection to a Database

You create an `OracleDataSource` using its constructor. You then open a connection to the database using `OracleDataSource.getConnection()`. The retrieved connection properties are derived from the `OracleDataSource` instance. See [Table 4–2, "Connection Properties Recognized by Oracle JDBC Drivers"](#) for the detailed list of connection properties. If you set the URL connection property, all other properties, including `TNSEntryName`, `DatabaseName`, `ServiceName`, `ServerName`, `PortNumber`, `Network Protocol`, and driver type are ignored. The syntax of the URL is discussed in [Chapter 3, "Datasources and URLs"](#)

Open a connection to the database using the JDBC `DataSource` class. To create a connection, you must specify a connection string containing a database URL.

### Specifying a Database URL, User Name, and Password

The following code sets the URL, user name, and password for a `datasource`:

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

(For URL format, see [Chapter 3, "Datasources and URLs"](#).)

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

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

---

**Note:** The username and password specified in the arguments override any username and password specified in the URL.

---

### Specifying a Database URL That Includes User Name and Password

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

```
String URL = "jdbc:oracle:oci:scott/tiger@myTNSEntry";
ods.setURL(URL);
Connection conn = ods.getConnection();
```

If you want to connect using the Thin driver you must specify the port number. 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 service identifier is `orcl`:

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

### Supported Connection Properties

Table 4–2 lists the connection properties that Oracle JDBC drivers support.

**Table 4–2 Connection Properties Recognized by Oracle JDBC Drivers**

Name	Type	Description
<code>accumulateBatchResult</code>	String (containing boolean value)	"true" causes the number of modified rows used to determine when to flush a batch accumulates across all batches flushed from a single statement. The default is "false", counting each batch separately
<code>database</code>	String	connect string for the database
<code>defaultBatchValue</code>	String (containing integer value)	default batch value that triggers an execution request (default value is "10")
<code>defaultExecuteBatch</code>	String (containing integer value)	default batch size when using Oracle batching
<code>defaultNchar</code>	String (containing boolean value)	"true" causes the default mode for all character data columns to be NCHAR.
<code>defaultRowPrefetch</code>	String (containing integer value)	default number of rows to prefetch from the server (default value is "10")

**Table 4-2 (Cont.) Connection Properties Recognized by Oracle JDBC Drivers**

<b>Name</b>	<b>Type</b>	<b>Description</b>
<code>disableDefineColumnType</code>	String (containing boolean value)	"true" causes <code>defineColumnType()</code> to have no effect.  This is highly recommended when using the Thin driver, especially when the database character set contains four byte characters that expand to two UCS2 surrogate characters, e.g. AL32UTF8. The method <code>defineColumnType()</code> provides no performance benefit (or any other benefit) when used with the 10g Release 1 (10.1) Thin driver. This property is provided so that you do not have to remove the calls from your code. This is especially valuable if you use the same code with Thin driver and either the OCI or Server Internal driver.
<code>DMSName</code>	String	name of the DMS Noun that is the parent of all JDBC DMS metrics. (see Note.)
<code>DMSType</code>	String	type of the DMS Noun that is the parent of all JDBC DMS metrics. (see Note.)
<code>fixedString</code>	String (containing boolean value)	"true" causes JDBC to use <code>FIXED CHAR</code> semantics when <code>setObject()</code> is called with a <code>String</code> argument. By default JDBC uses <code>VARCHAR</code> semantics. The difference is in blank padding. By default there is no blank padding. For example, 'a' does not equal 'a' in a <code>CHAR(4)</code> unless <code>fixedString</code> is "true".
<code>includeSynonyms</code>	String (containing boolean value)	"true" to include column information from predefined "synonym" SQL entities when you execute a <code>DataBaseMetaData.getColumns()</code> call; equivalent to <code>connection.setIncludeSynonyms()</code> call (default value is "false")
<code>internal_logon</code>	String	username used in an internal logon. Must be the role, such as <code>sysdba</code> or <code>sysoper</code> , that allows you to log on as <code>sys</code>
<code>oracle.jdbc.J2EE13Compliant</code>	String (containing boolean value)	"true" causes JDBC to use strict compliance for some edge cases. In general, Oracle's JDBC drivers allow some operations that are not permitted in the strict interpretation of J2EE 1.3. Setting this property to "true" will cause those cases to throw <code>SQLExceptions</code> . There are some other edge cases where Oracle's JDBC drivers have slightly different behavior than defined in J2EE 1.3. This results from Oracle having defined the behavior prior to the J2EE 1.3 specification and the resultant need for compatibility with existing customer code. Setting this property will result in full J2EE 1.3 compliance at the cost of incompatibility with some customer code. Can be either a system property or a connection property.

**Table 4-2 (Cont.) Connection Properties Recognized by Oracle JDBC Drivers**

Name	Type	Description
<code>oracle.jdbc.TcpNoDelay</code>	String (containing boolean value)	"true" causes the <code>TCP_NODELAY</code> property is set on the socket when using the Thin driver. See <code>java.net.SocketOptions.TCP_NODELAY</code> . Can be either a system property or a connection property.
<code>oracle.jdbc.ocinativelibrary</code>	String	name of the native library for the OCI driver. If not set, the default name, <code>libocijdbcX</code> ( <code>X</code> is the version number), is used.
<code>password</code>	String	the password for logging into the database
<code>processEscapes</code>	String (containing boolean value)	"true" if escape processing is enabled for all statements, "false" if escape processing is disabled (default value is "false")
<code>remarksReporting</code>	String (containing boolean value)	"true" if <code>getTables()</code> and <code>getColumns()</code> should report <code>TABLE_REMARKS</code> ; equivalent to using <code>setRemarksReporting()</code> (default value is "false")
<code>remarksReporting</code>	String (containing boolean value)	"true" causes <code>OracleDatabaseMetaData</code> to include remarks in the metadata. This can result in a substantial reduction in performance.
<code>restrictGetTables</code>	String (containing boolean value)	"true" causes JDBC to return a more refined value for <code>DatabaseMetaData.getTables()</code> . By default JDBC will return things that are not accessible tables. These can be non-table objects or accessible synonyms for inaccessible tables. If this property is "true", JDBC returns only accessible tables. This has a substantial performance penalty.
<code>server</code>	String	hostname of database
<code>useFetchSizeWithLongColumn</code>	String (containing boolean value)	"true" causes JDBC to prefetch rows even when there is a LONG or LONG RAW column in the result. By default JDBC fetches only one row at a time if there are LONG or LONG RAW columns in the result. Setting this property to true can improve performance but can also cause <code>SQLExceptions</code> if the results are too big.  We recommend avoiding LONG and LONG RAW columns; use LOB instead.
<code>user</code>	String	user name for logging into the database

See [Table 23-2, "OCI Driver Client Parameters for Encryption and Integrity"](#) and [Table 23-3, "Thin Driver Client Parameters for Encryption and Integrity"](#) for descriptions of encryption and integrity drivers.

### Using Roles for Sys Logon

To specify the role (mode) for `sys` logon, use the `internal_logon` connection property. (See [Table 4-2, "Connection Properties Recognized by Oracle JDBC Drivers"](#), for a

complete description of this connection property.) To logon as *sys*, set the `internal_logon` connection property to `sysdba` or `sysoper`.

---

---

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

---

---

For a bequeath connection, we can get a connection as "sys" by setting the `internal_logon` property. For a remote connection, we need additional password file setting procedures.

### Configuring To Permit Use of sysdba

Before the Thin driver can connect to the database as *sysdba*, you must configure the user as follows:

1. From the command line, type:

```
orapwd file=$ORACLE_HOME/dbs/orapw password=yourpass entries=5
```

2. In SQLPLUS, connect / as *sysdba*.

- To grant *sysdba* to a user *Username*, type:

```
grant SYSDBA to Username
```

- To grant *sysdba* to *sys*, type:

```
ALTER USER sys IDENTIFIED BY yourpass
```

3. Edit `init.ora` and add the line:

```
REMOTE_LOGIN_PASSWORDFILE=EXCLUSIVE
```

### Bequeath Connection and Sys Logon

The following example illustrates how to use the `internal_logon` and `sysdba` arguments to specify *sys* logon. This example works regardless of the database's national-language settings.

```
/** Example of bequeath connection **/  
import java.sql.*;  
import oracle.jdbc.*;  
import oracle.jdbc.pool.*;  
  
// create an OracleDataSource instance  
OracleDataSource ods = new OracleDataSource();  
  
// set necessary properties  
java.util.Properties prop = new java.util.Properties();  
prop.put("user", "sys");  
prop.put("password", "sys");  
prop.put("internal_logon", "sysdba");  
ods.setConnectionProperties(prop);  
  
// the url for bequeath connection  
String url = "jdbc:oracle:oci8:@";  
ods.setURL(url);
```

```
// retrieve the connection
Connection conn = ods.getConnection();
...
```

## Remote Connection

Password file pre-procedures are needed for getting connected to a remote database as user SYS, because the Oracle database security system requires a password file for remote connections as an administrator.

1. Set a password file on the server side, or on the remote database, using the password utility `orapwd`. You can add a password file for user `sys` as follows:

```
(UNIX) orapwd file=$ORACLE_HOME/dbs/orapw password=sys entries=200
(WINDOWS) orapwd file=$ORACLE_HOME\database\PWDsid_name.ora
password=sys entries=200
```

Please refer to the *Oracle Database Administrator's Guide* for its details. `file` must be the name of the password file. `password` is the password for the user `sys`. It can be altered using "alter user ..." in SQLPlus. You should set `entries` higher than the number of entries you expect.

The syntax for the password file name is different on Windows than on Unix.

2. Enable remote login as `sysdba`. This step grants `SYSDBA` and `SYSOPER` system privileges to individual users and lets them connect as themselves.

Stop the database. Then add the following line to (UNIX) `init_service_name.ora` (Windows) `init.ora`:

```
remote_login_passwordfile=exclusive
```

The `init_service_name.ora` file is located at `ORACLE_HOME/dbs/` and also at `ORACLE_HOME/admin/db_name/pfile/`. Keep the two files synchronized.

The `init.ora` file is located at `%ORACLE_BASE%\ADMIN\db_name\pfile\`.

3. (Optional) Change the password for the `sys` user

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

4. Verify whether `sys` has the `sysdba` privilege. The following message should come up:

```
SQL> select * from v$pwfile_users;
USERNAME          SYSDB  SYSOP
-----
SYS                TRUE   TRUE
```

5. Restart the remote database.

**Example 4-1 Using sys Logon To Make a Remote Connection**

This example works regardless of database's language settings

```
/** case of remote connection using sys */
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
// create an OracleDataSource
OracleDataSource ods = new OracleDataSource();
// set connection properties
java.util.Properties prop = new java.util.Properties();
prop.put("user", "sys");
prop.put("password", "sys");
prop.put("internal_logon", "sysoper");
ods.setConnectionProperties(prop);
// set the url
// the url can use oci driver as well as:
// url = "jdbc:oracle:oci8:@inst1"; the inst1 is a remote database
String url = "jdbc:oracle:thin:@//myHost:1521/service_name";
ods.setURL(url);
// get the connection
Connection conn = ods.getConnection();
```

**Properties for Oracle Performance Extensions**

Some of these properties are for use with Oracle performance extensions. Setting these properties is equivalent to using corresponding methods on the `OracleConnection` object, as follows:

- Setting the `defaultRowPrefetch` property is equivalent to calling `setDefaultRowPrefetch()`.  
See ["Oracle Row Prefetching"](#) on page 22-15.
- Setting the `remarksReporting` property is equivalent to calling `setRemarksReporting()`.  
See ["Database MetaData TABLE\\_REMARKS Reporting"](#) on page 22-20.
- Setting the `defaultBatchValue` property is equivalent to calling `setDefaultExecuteBatch()`.  
See ["Oracle Update Batching"](#) on page 22-3.

**Example** The following example shows how to use the `put()` method of the `java.util.Properties` class, in this case to set Oracle performance extension parameters.

```
//import packages and register the driver
import java.sql.*;
import java.math.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;

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

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

## 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 a `JDBC ResultSet` object.

To continue the example, once you create the `Statement` object `stmt`, the next step is to execute a query that populates a `ResultSet` object with the contents of the `ENAME` (employee name) column of a table of employees named `EMP`:

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

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

## Processing the Result Set

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

To pull data out of the result set as you iterate through it, use the appropriate `getXXX()` methods of the `ResultSet` object, where `XXX` corresponds to a Java datatype.

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

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

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

## 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 both the result set and the statement releases the corresponding cursor in the database; if you close only the result set, the cursor is not released.

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

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

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

---

---

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

---

---

## Making Changes to the Database

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

Use the `setXXX()` methods on the `PreparedStatement` object to bind data into the prepared statement to be sent to the database. The various `setXXX()` methods are described in "[The setObject\(\) and setOracleObject\(\) Methods](#)" on page 11-9 and "[Other setXXX\(\) Methods](#)" on page 11-9.

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

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

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

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

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

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

---

## Committing Changes

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

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

(For further discussion of auto-commit mode and an example of disabling it, see ["Disabling Auto-Commit Mode"](#) on page 26-4.)

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

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

or:

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

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

---

---

### Important:

- If auto-commit mode is disabled and you close the connection without explicitly committing or rolling back your last changes, then an implicit `COMMIT` operation is executed.
  - Any DDL operation, such as `CREATE` or `ALTER`, always includes an implicit `COMMIT`. If auto-commit mode is disabled, this implicit `COMMIT` will not only commit the DDL statement, but also any pending DML operations that had not yet been explicitly committed or rolled back.
- 
- 

## Closing the Connection

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

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

---

---

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

---

---

## Sample: Connecting, Querying, and Processing the Results

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

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

```
import java.sql.*;
import java.math.*;
import java.io.*;
import java.awt.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;

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

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

        //close the result set, statement, and the connection
        rset.close();
        stmt.close();
        conn.close();
    }
}
```

If you want to adapt the code for the OCI driver, replace the `OracleDataSource.setURL()` invocation with the following:

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

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

## Datatype Mappings

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

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

### Table of Mappings

For reference, [Table 4–3](#) shows the default mappings between SQL datatypes, JDBC typecodes, standard Java types, and Oracle extended types.

The **SQL Datatypes** column lists the SQL types that exist in the 10g Release 1 (10.1) database.

The **JDBC Typecodes** column lists data typecodes supported by the JDBC standard and defined in the `java.sql.Types` class, or by Oracle in the `oracle.jdbc.OracleTypes` class. For standard typecodes, the codes are identical in these two classes.

The **Standard Java Types** column lists standard types defined in the Java language.

The **Oracle Extension Java Types** column lists the `oracle.sql.*` Java types that correspond to each SQL datatype in the database. These are Oracle extensions that let you retrieve all SQL data in the form of a `oracle.sql.*` Java type. Mapping SQL datatypes into the `oracle.sql` datatypes lets you store and retrieve data without losing information. Refer to "Package `oracle.sql`" on page 10-5 for more information on the `oracle.sql.*` package.

**Table 4-3 Default Mappings Between SQL Types and Java Types**

SQL Datatypes	JDBC Typecodes	Standard Java Types	Oracle Extension Java Types
<b>STANDARD JDBC 1.0 TYPES:</b>			
CHAR	<code>java.sql.Types.CHAR</code>	<code>java.lang.String</code>	<code>oracle.sql.CHAR</code>
VARCHAR2	<code>java.sql.Types.VARCHAR</code>	<code>java.lang.String</code>	<code>oracle.sql.CHAR</code>
LONG	<code>java.sql.Types.LONGVARCHAR</code>	<code>java.lang.String</code>	<code>oracle.sql.CHAR</code>
NUMBER	<code>java.sql.Types.NUMERIC</code>	<code>java.math.BigDecimal</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.DECIMAL</code>	<code>java.math.BigDecimal</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.BIT</code>	<code>boolean</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.TINYINT</code>	<code>byte</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.SMALLINT</code>	<code>short</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.INTEGER</code>	<code>int</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.BIGINT</code>	<code>long</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.REAL</code>	<code>float</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.FLOAT</code>	<code>double</code>	<code>oracle.sql.NUMBER</code>
NUMBER	<code>java.sql.Types.DOUBLE</code>	<code>double</code>	<code>oracle.sql.NUMBER</code>
RAW	<code>java.sql.Types.BINARY</code>	<code>byte[]</code>	<code>oracle.sql.RAW</code>
RAW	<code>java.sql.Types.VARBINARY</code>	<code>byte[]</code>	<code>oracle.sql.RAW</code>
LONGRAW	<code>java.sql.Types.LONGVARBINARY</code>	<code>byte[]</code>	<code>oracle.sql.RAW</code>
DATE	<code>java.sql.Types.DATE</code>	<code>java.sql.Date</code>	<code>oracle.sql.DATE</code>
DATE	<code>java.sql.Types.TIME</code>	<code>java.sql.Time</code>	<code>oracle.sql.DATE</code>
TIMESTAMP	<code>java.sql.Types.TIMESTAMP</code>	<code>java.sql.Timestamp</code>	<code>oracle.sql.TIMESTAMP</code> (see Note)
<b>STANDARD JDBC 2.0 TYPES:</b>			
BLOB	<code>java.sql.Types.BLOB</code>	<code>java.sql.Blob</code>	<code>oracle.sql.BLOB</code>
CLOB	<code>java.sql.Types.CLOB</code>	<code>java.sql.Clob</code>	<code>oracle.sql.CLOB</code>
user-defined object	<code>java.sql.Types.STRUCT</code>	<code>java.sql.Struct</code>	<code>oracle.sql.STRUCT</code>

**Table 4-3 (Cont.) Default Mappings Between SQL Types and Java Types**

SQL Datatypes	JDBC Typecodes	Standard Java Types	Oracle Extension Java Types
user-defined reference	java.sql.Types.REF	java.sql.Ref	oracle.sql.REF
user-defined collection	java.sql.Types.ARRAY	java.sql.Array	oracle.sql.ARRAY
<b>ORACLE EXTENSIONS:</b>			
BFILE	oracle.jdbc.OracleTypes.BFILE	n/a	oracle.sql.BFILE
ROWID	oracle.jdbc.OracleTypes.ROWID	n/a	oracle.sql.ROWID
REF CURSOR type	oracle.jdbc.OracleTypes.CURSOR	java.sql.ResultSet	oracle.jdbc.OracleResultSet
TIMESTAMP	oracle.jdbc.OracleTypes.TIMESTAMP	java.sql.Timestamp	oracle.sql.TIMESTAMP
TIMESTAMP WITH TIME ZONE	oracle.jdbc.OracleTypes.TIMESTAMP_TZ	java.sql.Timestamp	oracle.sql.TIMESTAMP_TZ
TIMESTAMP WITH LOCAL TIME ZONE	oracle.jdbc.OracleTypes.TIMESTAMP_PLTZ	java.sql.Timestamp	oracle.sql.TIMESTAMP_PLTZ

---

**Note:** For database versions, such as 8.1.7, that do not support the `TIMESTAMP` datatype, this is mapped to `DATE`.

---

For a list of all the Java datatypes to which you can validly map a SQL datatype, see "[Valid SQL-JDBC Datatype Mappings](#)" on page 24-1.

See [Chapter 10, "Oracle Extensions"](#), for more information on type mappings. In [Chapter 10](#) you can also find more information on the following:

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

## Notes Regarding Mappings

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

### Regarding User-Defined Types

User-defined types such as objects, object references, and collections map by default to weak Java types (such as `java.sql.Struct`), but alternatively can map to strongly typed *custom Java classes*. Custom Java classes can implement one of two interfaces:

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

For information about custom Java classes and the `SQLData` and `ORAData` interfaces, see "[Mapping Oracle Objects](#)" on page 13-1 and "[Creating and Using Custom Object](#)"

[Classes for Oracle Objects](#)" on page 13-7. (Although these sections focus on custom Java classes for user-defined objects, there is some general information about other kinds of custom Java classes as well.)

### Regarding NUMBER Types

For the different typecodes that an Oracle `NUMBER` value can correspond to, call the getter routine that is appropriate for the size of the data for mapping to work properly. For example, call `getBytes()` to get a Java `tinyint` value, for an item `x` where  $-128 < x < 128$ .

## Java Streams in JDBC

This section covers the following topics:

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

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

Oracle JDBC drivers support the manipulation of data streams in either direction between server and client. The drivers support all stream conversions: binary, ASCII, and Unicode. Following is a brief description of each type of stream:

- **binary stream**—Used for `RAW` bytes of data. This corresponds to the `getBinaryStream()` method.
- **ASCII stream**—Used for ASCII bytes in ISO-Latin-1 encoding. This corresponds to the `getAsciiStream()` method.
- **Unicode stream**—Used for Unicode bytes with the UTF-16 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 14, "Working with LOBs and BFILES"](#).

### Streaming LONG or LONG RAW Columns

When a query selects one or more `LONG` or `LONG RAW` columns, the JDBC driver transfers these columns to the client in streaming mode. 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 conversions for you, depending on the character set of your database and the

driver. For more information about globalization support, see ["JDBC Methods Dependent On Conversion"](#) on page 12-3.

---

---

**Note:** Do not create tables with `LONG` columns. Use `LOB` columns (`CLOB`, `NCLOB`, `BLOB`) instead. `LONG` columns are supported only for backward compatibility. Oracle Corporation also recommends that you convert existing `LONG` columns to `LOB` columns. `LOB` columns are subject to far fewer restrictions than `LONG` columns. Further, `LOB` functionality is enhanced in every release, whereas `LONG` functionality has been static for several releases.

---

---

### 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.

### LONG Data Conversions

When you get `LONG` data with `getAsciiStream()`, the drivers assume that the underlying data in the database uses an `US7ASCII` or `WE8ISO8859P1` character set. If the assumption is true, the drivers return bytes corresponding to ASCII characters. If the database is not using an `US7ASCII` or `WE8ISO8859P1` character set, a call to `getAsciiStream()` returns meaningless information.

When you get `LONG` data with `getUnicodeStream()`, you get a stream of Unicode characters in the UTF-16 encoding. This applies to all underlying database character sets that Oracle supports.

When you get `LONG` data with `getBinaryStream()`, there are two possible cases:

- If the driver is JDBC OCI and the client character set is not `US7ASCII` or `WE8ISO8859P1`, then a call to `getBinaryStream()` returns UTF-8. If the client character set is `US7ASCII` or `WE8ISO8859P1`, then the call returns a `US7ASCII` stream of bytes.
- If the driver is JDBC Thin and the database character set is not `US7ASCII` or `WE8ISO8859P1`, then a call to `getBinaryStream()` returns UTF-8. If the server-side character set is `US7ASCII` or `WE8ISO8859P1`, then the call returns a `US7ASCII` stream of bytes.

For more information on how the drivers return data based on character set, see [Chapter 12, "Globalization Support"](#).

---

---

**Note:** Receiving `LONG` or `LONG RAW` columns as a stream (the default case) requires you to pay special attention to the order in which you receive data from the database. For more information, see ["Data Streaming and Multiple Columns"](#) on page 4-19.

---

---

Table 4-4 summarizes LONG and LONG RAW data conversions for each stream type.

**Table 4-4 LONG and LONG RAW Data Conversions**

Datatype	BinaryStream	AsciiStream	UnicodeStream
LONG	bytes representing characters in Unicode UTF-8. The bytes can represent characters in US7ASCII or WE8ISO8859P1 if: <ul style="list-style-type: none"> <li>the database character set is US7ASCII or WE8ISO8859P1.</li> </ul>	bytes representing characters in ISO-Latin-1 (WE8ISO8859P1) encoding	bytes representing characters in Unicode UTF-16 encoding
LONG RAW	as-is	ASCII representation of hexadecimal bytes	Unicode representation of hexadecimal bytes

### Streaming Example for LONG RAW Data

One of the features of a `getXXXStream()` method is that it allows you to fetch data incrementally. In contrast, `getBytes()` fetches all the data in one call. This section contains two examples of getting a stream of binary data. The first version uses the `getBinaryStream()` method to obtain LONG RAW data; the second version uses the `getBytes()` method.

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

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

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

The following Java code snippet writes the data from the LESLIE LONG RAW column into a file called `leslie.gif`:

```
ResultSet rset = stmt.executeQuery
    ("select GIFDATA from streamexample where NAME='LESLIE'");

// get first row
if (rset.next())
{
    // Get the GIF data as a stream from Oracle to the client
    InputStream gif_data = rset.getBinaryStream (1);
    try
    {
        FileOutputStream file = null;
        file = new FileOutputStream ("leslie.gif");
        int chunk;
        while ((chunk = gif_data.read()) != -1)
            file.write(chunk);
    }
    catch (Exception e)
    {
        String err = e.toString();
        System.out.println(err);
    }
}
```

```
        finally
        {
            if file != null()
                file.close();
        }
    }
}
```

In this example the contents of the `GIFDATA` column are transferred incrementally in chunk-sized pieces between the database and the client. The `InputStream` object returned by the call to `getBinaryStream()` reads the data directly from the database connection.

**Getting a LONG RAW Data Column with `getBytes()`** This version of the example gets the content of the `GIFDATA` column with `getBytes()` instead of `getBinaryStream()`. In this case, the driver fetches all the data in one call and stores it in a byte array. The previous code snippet can be rewritten as:

```
ResultSet rset2 = stmt.executeQuery
    ("select GIFDATA from streamexample where NAME='LESLIE'");

// get first row
if (rset2.next())
{
    // Get the GIF data as a stream from Oracle to the client
    byte[] bytes = rset2.getBytes(1);
    try
    {
        FileOutputStream file = null;
        file = new FileOutputStream ("leslie2.gif");
        file.write(bytes);
    }
    catch (Exception e)
    {
        String err = e.toString();
        System.out.println(err);
    }
    finally
    {
        if file != null()
            file.close();
    }
}
```

Because a `LONG RAW` column can contain up to 2 gigabytes of data, the `getBytes()` example will probably use much more memory than the `getBinaryStream()` example. Use streams if you do not know the maximum size of the data in your `LONG` or `LONG RAW` columns.

### Avoiding Streaming for `LONG` or `LONG RAW`

The JDBC driver automatically streams any `LONG` and `LONG RAW` columns. However, there may be situations where you want to avoid data streaming. For example, if you have a very small `LONG` column, you might want to avoid returning the data incrementally and instead, return the data in one call.

To avoid streaming, use the `defineColumnType()` method to redefine the type of the `LONG` column. For example, if you redefine the `LONG` or `LONG RAW` column as type `VARCHAR` or `VARBINARY`, then the driver will not automatically stream the data.

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

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

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

```
//cast the statement stmt to an OracleStatement
oracle.jdbc.OracleStatement ostmt =
    (oracle.jdbc.OracleStatement)stmt;

//redefine the LONG column at index position 1 to VARBINARY
ostmt.defineColumnType(1, Types.VARBINARY);

// Do a query to get the images named 'LESLIE'
ResultSet rset = ostmt.executeQuery
    ("select GIFDATA from streamexample where NAME='LESLIE'");

// The data is not streamed here
rset.next();
byte [] bytes = rset.getBytes(1);
```

## Streaming CHAR, VARCHAR, or RAW Columns

If you use the `defineColumnType()` Oracle extension to redefine a `CHAR`, `VARCHAR`, or `RAW` column as a `LONGVARCHAR` or `LONGVARBINARY`, then you can get the column as a stream. The program will behave as if the column were actually of type `LONG` or `LONG RAW`. Note that there is not much point to this, because these columns are usually short.

If you try to get a `CHAR`, `VARCHAR`, or `RAW` column as a data stream without redefining the column type, the JDBC driver will return a `Java InputStream`, but no real streaming occurs. In the case of these datatypes, the JDBC driver fully fetches the data into an in-memory buffer during a call to the `executeQuery()` method or `next()` method. The `getXXXStream()` entry points return a stream that reads data from this buffer.

## Data Streaming and Multiple Columns

If your query selects multiple columns and one of the columns contains a data stream, then the contents of the columns following the stream column are not available until the stream has been read, and the stream column is no longer available once any following column is read. Any attempt to read a column beyond a streaming column closes the streaming column. See "[Streaming Data Precautions](#)" on page 4-22 for more information.

### Streaming Example with Multiple Columns

Consider the following query:

```
ResultSet rset = stmt.executeQuery
    ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
    //get the date data
    java.sql.Date date = rset.getDate(1);

    // get the streaming data
    InputStream is = rset.getAsciiStream(2);

    // Open a file to store the gif data
    FileOutputStream file = new FileOutputStream ("ascii.dat");

    // Loop, reading from the ascii stream and
    // write to the file
    int chunk;
    while ((chunk = is.read ()) != -1)
        file.write(chunk);
    // Close the file
    file.close();

    //get the number column data
    int n = rset.getInt(3);
}
```

The incoming data for each row has the following shape:

```
<a date><the characters of the long column><a number>
```

As you process each row of the iterator, you must complete any processing of the stream column before reading the number column.

An exception to this behavior is LOB data, which is also transferred between server and client as a Java stream. For more information on how the driver treats LOB data, see "[Streaming LOBs and External Files](#)" on page 4-21.

### Bypassing Streaming Data Columns

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

In the following example, the stream data in the `LONG` column is discarded and the data from only the `DATE` and `NUMBER` column is recovered:

```
ResultSet rset = stmt.executeQuery
    ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");

while rset.next()
{
    //get the date
    java.sql.Date date = rset.getDate(1);

    // access the stream data and discard it with close()
    InputStream is = rset.getAsciiStream(2);
```

```
is.close();

// get the number column data
int n = rset.getInt(3);
}
```

## Streaming LOBs and External Files

The term *large object* (LOB) refers to a data item that is too large to be stored directly in a database table. Instead, a locator is stored in the database table and points to the location of the actual data. External files (binary files, or BFILEs) are managed similarly. The JDBC drivers can support these types through the use of streams:

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

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

### Streaming BLOBs and CLOBs

When a query selects one or more CLOB or BLOB columns, the JDBC driver transfers to the client the data pointed to by the locator. The driver performs the transfer as a Java stream. To manipulate CLOB or BLOB data from JDBC, use methods in the Oracle extension classes `oracle.sql.BLOB` and `oracle.sql.CLOB`. These classes provide functionality such as reading from the CLOB or BLOB into an input stream, writing from an output stream into a CLOB or BLOB, determining the length of a CLOB or BLOB, and closing a CLOB or BLOB.

For a complete discussion of how to use streaming CLOB and BLOB data, see ["Reading and Writing BLOB and CLOB Data"](#) on page 14-4. CLOB and BLOB data may also be streamed with the same mechanism as for LONG and LONG RAW. See ["Shortcuts For Inserting and Retrieving CLOB Data"](#) on page 14-12.

### Streaming BFILEs

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

When a query selects one or more BFILE columns, the JDBC driver transfers to the client the file pointed to by the locator. The transfer is performed in a Java stream. To manipulate BFILE data from JDBC, use methods in the Oracle extension class `oracle.sql.BFILE`. This class provides functionality such as reading from the BFILE into an input stream, writing from an output stream into a BFILE, determining the length of a BFILE, and closing a BFILE.

For a complete discussion of how to use streaming BFILE data, see ["Reading BFILE Data"](#) on page 14-16.

## 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 4-20. For information on how to avoid closing a stream and discarding its data by accident, see ["Streaming Data Precautions"](#) on page 4-22.

## Notes and Precautions on Streams

This section discusses several noteworthy and cautionary issues regarding the use of streams:

- [Streaming Data Precautions](#)
- [Using Streams to Avoid Limits on `setBytes\(\)` and `setString\(\)`](#)
- [Streaming and Row Prefetching](#)

### Streaming Data Precautions

This section describes some of the precautions you must take to ensure that you do not accidentally discard or lose your stream data. The drivers automatically discard stream data if you perform any JDBC operation that communicates with the database, other than reading the current stream. Two common precautions are described:

- Use the stream data after you access it.

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

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

If your query selects multiple columns, the database sends each row as a set of bytes representing the columns in the `SELECT` order. If one of the columns contains stream data, the database sends the entire data stream before proceeding to the next column.

If you do not use the `SELECT`-list order to access data, then you can lose the stream data. That is, if you bypass the stream data column and access data in a column that follows it, the stream data will be lost. For example, if you try to access the data for the `NUMBER` column *before* reading the data from the stream data column, the JDBC driver first reads then discards the streaming data automatically. This can be very inefficient if the `LONG` column contains a large amount of data.

If you try to access the `LONG` column later in the program, the data will not be available and the driver will return a "Stream Closed" error.

The second point is illustrated in the following example:

```
ResultSet rset = stmt.executeQuery
    ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
    int n = rset.getInt(3); // This discards the streaming data
    InputStream is = rset.getAsciiStream(2);
                        // Raises an error: stream closed.
}
```

If you get the stream but do not use it *before* you get the `NUMBER` column, the stream still closes automatically:

```

ResultSet rset = stmt.executeQuery
    ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
    InputStream is = rset.getAsciiStream(2); // Get the stream
    int n = rset.getInt(3);
    // Discards streaming data and closes the stream
}
int c = is.read(); // c is -1: no more characters to read-stream closed

```

### Using Streams to Avoid Limits on `setBytes()` and `setString()`

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

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

**Table 4-5 Bind-Size Limitations By**

Database Version	maximum <code>setBytes()</code> (equals maximum RAW size)	maximum <code>setString()</code> (equals maximum VARCHAR2 size)
Oracle8 and later	2000	4000
Oracle7	255	2000

**Note:** This discussion applies to binds in SQL, not PL/SQL. If you use `setBinaryStream()` in PL/SQL, the maximum array size is 32 Kbytes -7.

### Streaming and Row Prefetching

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

Row prefetching is an Oracle performance enhancement that allows multiple rows of data to be retrieved with each trip to the database. See "[Oracle Row Prefetching](#)" on page 22-15.

## Stored Procedure Calls in JDBC Programs

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

- [PL/SQL Stored Procedures](#)
- [Java Stored Procedures](#)

### PL/SQL Stored Procedures

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

```
// SQL92 syntax
CallableStatement cs1 = conn.prepareStatement
    ( "{call proc (?,?)}" ); // stored proc
CallableStatement cs2 = conn.prepareStatement
    ( "{? = call func (?,?)}" ); // stored func
// Oracle PL/SQL block syntax
CallableStatement cs3 = conn.prepareStatement
    ( "begin proc (?,?); end;" ); // stored proc
CallableStatement cs4 = conn.prepareStatement
    ( "begin ? := func(?,?); end;" ); // stored func
```

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

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

The function invocation in your JDBC program should look like:

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

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

## Java Stored Procedures

You can use JDBC to invoke Java stored procedures through the SQL and PL/SQL engines. The syntax for calling Java stored procedures is the same as the syntax for calling PL/SQL stored procedures, presuming they have been properly "published" (that is, have had call specifications written to publish them to the Oracle data dictionary).

## Processing SQL Exceptions

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

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

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

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

## Retrieving Error Information

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

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

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

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

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

```
exception: Invalid column type
```

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

---

---

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

---

---

## Printing the Stack Trace

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

- `printStackTrace()`

This method prints the stack trace of the throwable object to the standard error stream. You can also specify a `java.io.PrintStream` object or `java.io.PrintWriter` object for output.

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

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

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

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

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

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

```
java.sql.SQLException: Invalid column index
at oracle.jdbc.dbaccess.DBError.check_error(DBError.java:235)
at oracle.jdbc.OracleStatement.prepare_for_new_get(OracleStatement.java:1560)
at oracle.jdbc.OracleStatement.getStringValue(OracleStatement.java:1653)
at oracle.jdbc.OracleResultSet.getString(OracleResultSet.java:175)
at Employee.main(Employee.java:41)
```