Using Java Database Connectivity (JDBC) with Oracle

By Bulusu Lakshman

Apr 5, 2002

This chapter is from the book

Oracle and Java Development

Learn More Buy

Fundamental Steps in JDBC

The fundamental steps involved in the process of connecting to a database and executing a query consist of the following:

- Import JDBC packages.
- Load and register the JDBC driver.
- Open a connection to the database.
- Create a statement object to perform a query.
- Execute the statement object and return a query resultset.
- Process the resultset.
- Close the resultset and statement objects.
- Close the connection.

These steps are described in detail in the sections that follow.

Import JDBC Packages

This is for making the JDBC API classes immediately available to the application program. The following import statement should be included in the program irrespective of the JDBC driver being used:

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

Additionally, depending on the features being used, Oracle-supplied JDBC packages might need to be imported. For example, the following packages might need to be imported while using the

Oracle extensions to JDBC such as using advanced data types such as BLOB, and so on.

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

Load and Register the JDBC Driver

This is for establishing a communication between the JDBC program and the Oracle database. This is done by using the static `registerDriver()` method of the `DriverManager` class of the JDBC API. The following line of code does this job:

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

JDBC Driver Registration

For the entire Java application, the JDBC driver is registered only once per each database that needs to be accessed. This is true even when there are multiple database connections to the same data server.

Alternatively, the `forName()` method of the `java.lang.Class` class can be used to load and register the JDBC driver:

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

However, the `forName()` method is valid for only JDK-compliant Java Virtual Machines and implicitly creates an instance of the Oracle driver, whereas the `registerDriver()` method does this explicitly.

Connecting to a Database

Once the required packages have been imported and the Oracle JDBC driver has been loaded and registered, a database connection must be established. This is done by using the `getConnection()` method of the `DriverManager` class. A call to this method creates an object instance of the `java.sql.Connection` class. The `getConnection()` requires three input parameters, namely, a connect string, a username, and a password. The connect string should specify the JDBC driver to be yes and the database instance to connect to.

The `getConnection()` method is an overloaded method that takes

- Three parameters, one each for the URL, username, and password.
- Only one parameter for the database URL. In this case, the URL contains the username and password.

The following lines of code illustrate using the `getConnection()` method:

```
Connection conn = DriverManager.getConnection(URL, username, passwd);
Connection conn = DriverManager.getConnection(URL);
```

where URL, username, and passwd are of `String` data types.

We will discuss the methods of opening a connection using the Oracle JDBC OCI and thin _drivers.

When using the OCI driver, the database can be specified using the TNSNAMES entry in the tnsnames.ora file. For example, to connect to a database on a particular host as user oratest and password oratest that has a TNSNAMES entry of oracle.world, use the following code:

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

Both the ":" and "@" are mandatory.

When using the JDBC thin driver, the TNSNAMES entry cannot be used to identify the database. There are two ways of specifying the connect string in this case, namely,

- Explicitly specifying the hostname, the TCP/IP port number, and the Oracle SID of the database to connect to. This is for thin driver only.
- Specify a Net8 keyword-value pair list.

For example, for the explicit method, use the following code to connect to a database on host `training` where the TCP/IP listener is on port 1521, the SID for the database instance is Oracle, the username and password are both oratest:

```
Connection conn = DriverManager.getConnection
                ("jdbc:oracle:thin:@training:1521:Oracle",
 "oratest", "oratest");
```

For the Net8 keyword-value pair list, use the following:

```
Connection conn = DriverManager.getConnection
           ("jdbc:oracle:thin@(description=(address=
           (host=training)(protocol=tcp)(port=1521))
           (connect_data=(sid=Oracle))) ", _"oratest", "oratest");
```

This method can also be used for the JDBC OCI driver. Just specify `oci8` instead of `thin` in the above keyword-value pair list.

Querying the Database

Querying the database involves two steps: first, creating a statement object to perform a query, and second, executing the query and returning a resultset.

Creating a Statement Object

This is to instantiate objects that run the query against the database connected to. This is done by the `createStatement()` method of the `conn Connection` object created above. A call to this method creates an object instance of the `Statement` class. The following line of code illustrates this:

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

Executing the Query and Returning a ResultSet

Once a `Statement` object has been constructed, the next step is to execute the query. This is done by using the `executeQuery()` method of the `Statement` object. A call to this method takes as parameter a SQL SELECT statement and returns a JDBC `ResultSet` object. The following line of code illustrates this using the `sql_stmt` object created above:

```
ResultSet rset = sql_stmt.executeQuery
      ("SELECT empno, ename, sal, deptno FROM emp ORDER BY ename");
```

Alternatively, the SQL statement can be placed in a string and then this string passed to the executeQuery() function. This is shown below.

```
String sql = "SELECT empno, ename, sal, deptno FROM emp ORDER BY ename";
ResultSet rset = sql_stmt.executeQuery(sql);
```

Statement and ResultSet Objects

Statement and ResultSet objects open a corresponding cursor in the database for SELECT and other DML statements.

The above statement executes the SELECT statement specified in between the double quotes and stores the resulting rows in an instance of the `ResultSet` object named `rset`.

Processing the Results of a Database Query That Returns Multiple Rows

Once the query has been executed, there are two steps to be carried out:

- Processing the output resultset to fetch the rows
- Retrieving the column values of the current row

The first step is done using the `next()` method of the `ResultSet` object. A call to `next()` is executed in a loop to fetch the rows one row at a time, with each call to `next()` advancing the control to the next available row. The `next()` method returns the Boolean value true while rows are still available for fetching and returns false when all the rows have been fetched.

The second step is done by using the `getXXX()` methods of the JDBC rset object. Here `getXXX()` corresponds to the `getInt()`, `getString()` etc with XXX being replaced by a Java datatype.

The following code demonstrates the above steps:

```
String str;
while (rset.next())
 {
 str = rset.getInt(1)+ " "+ rset.getString(2)+ "
        "+rset.getFloat(3)+ " "rset.getInt(4)+ "\n";
 }
byte buf[] = str.getBytes();
OutputStream fp = new FileOutputStream("query1.lst");
fp.write(buf);
fp.close();
```

Here the 1, 2, 3, and 4 in `rset.getInt()`, `rset.getString()`, `getFloat()`, and `getInt()` respectively denote the position of the columns in the SELECT statement, that is, the first column `empno`, second column `ename`, third column `sal`, and fourth column `deptno` of the SELECT statement respectively.

Specifying `get()` Parameters

The parameters for the `getXXX()` methods can be specified by position of the corresponding columns as numbers 1, 2, and so on, or by directly specifying the column names enclosed in double quotes, as `getString("ename")` and so on, or a combination of both.

Closing the ResultSet and Statement

Once the `ResultSet` and `Statement` objects have been used, they must be closed explicitly. This is done by calls to the `close()` method of the `ResultSet` and `Statement` classes. The following code illustrates this:

```
rset.close();
sql_stmt.close();
```

If not closed explicitly, there are two disadvantages:

- Memory leaks can occur
- Maximum Open cursors can be exceeded

Closing the `ResultSet` and `Statement` objects frees the corresponding cursor in the database.

Closing the Connection

The last step is to close the database connection opened in the beginning after importing the packages and loading the JDBC drivers. This is done by a call to the `close()` method of the Connection class.

The following line of code does this:

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

Explicitly Close your Connection

Closing the `ResultSet` and `Statement` objects does not close the connection. The connection should be closed by explicitly invoking the `close()` method of the `Connection` class.

A complete example of the above procedures using a JDBC thin driver is given below. This program queries the emp table and writes the output rows to an operating system file.

```
//Import JDBC package
import java.sql.*;
// Import Java package for File I/O
import java.io.*;
public class QueryExample {
 public static void main (String[] args) throws SQLException, IOException
{
  //Load and register Oracle driver
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
 //Establish a connection
 Connection conn =  DriverManager.getConnection("jdbc:oracle:thin:
 @training:1521:Oracle", "oratest", "oratest");
 //Create a Statement object
 Statement sql_stmt = conn.createStatement();
 //Create a ResultSet object, execute the query and return a
 // resultset
 ResultSet rset = sql_stmt.executeQuery("SELECT empno, ename, sal,
 deptno FROM emp ORDER BY ename");
 //Process the resultset, retrieve data in each row, column by column
 //and write to an operating system file
String str = "";
while (rset.next())
 {
 str += rset.getInt(1)+" "+ rset.getString(2)+" "+
 rset.getFloat(3)+" "+rset.getInt(4)+"\n";
 }
 byte buf[] = str.getBytes();
OutputStream fp = new FileOutputStream("query1.lst");
fp.write(buf);
fp.close();
//Close the ResultSet and Statement
 rset.close();
 sql_stmt.close();
 //Close the database connection
 conn.close();
 }
}
```

Processing the Results of a Database Query That Returns a Single Row

The above sections and the complete example explained the processing of a query that returned multiple rows. This section highlights the processing of a single-row query and explains how to write code that is the analogue of the PL/SQL exception NO_DATA_FOUND.

NO DATA FOUND Exception

NO_DATA_FOUND exception in PL/SQL is simulated in JDBC by using the return value of the `next()` method of the `ResultSet` object. A value of `false` returned by the `next()` method identifies a NO_DATA_FOUND exception.

Consider the following code (this uses the `ResultSet` object `rset` defined in the above sections):

```
if (rset.next())
 // Process the row returned
else
 System.out.println("The Employee with Empno "+ args[1] +
                    "does not exist");
```

Instead of the while loop used earlier, an if statement is used to determine whether the SELECT statement returned a row or not.

Datatype Mappings

Corresponding to each SQL data type, there exist mappings to the corresponding JDBC Types, standard Java types, and the Java types provided by Oracle extensions. These are required to be used in JDBC programs that manipulate data and data structures based on these types.

There are four categories of Data types any of which can be mapped to the others. These are:

- **SQL Data types**—These are Oracle SQL data types that exist in the database.
- **JDBC Typecodes**—These are the data typecodes supported by JDBC as defined in the `java.sql.Types` class or defined by Oracle in `oracle.jdbc.driver.OracleTypes` class.
- **Java Types**—These are the standard types defined in the Java language.

- **Oracle Extension Java Types**—These are the Oracle extensions to the SQL data types and are defined in the `oracle.sql.*` class. Mapping SQL data types to the `oracle.sql.*` Java types enables storage and retrieval of SQL data without first converting into Java format thus preventing any loss of information.

Table 3.1 lists the default mappings existing between these four different types.

Table 3.1 Standard and Oracle-specific SQL-Java Data Type Mappings

SQL Data types	JDBC Type codes	Standard Java Types	Oracle Extension Java _ Types
Standard JDBC 1.0 Types			
CHAR	java.sql.Types.CHAR	java.lang.String	oracle.sql.CHAR
VARCHAR2	java.sql.Types.VARCHAR	java.lang.String	oracle.sql.CHAR
LONG	java.sql.Types. LONGVARCHAR	java.lang.String	oracle.sql.CHAR_
NUMBER	java.sql.Types.NUMERIC	java.math.BigDecimal	oracle.sql.NUMBER
NUMBER	java.sql.Types.DECIMAL	java.math.BigDecimal	oracle.sql.NUMBER
NUMBER	java.sql.Types.BIT	Boolean	oracle.sql.NUMBER
NUMBER	java.sql.Types.TINYINT	byte	oracle.sql.NUMBER
NUMBER	java.sql.Types.SMALLINT	short	oracle.sql.NUMBER
NUMBER	java.sql.Types.INTEGER	int	oracle.sql.NUMBER
NUMBER	java.sql.Types.BIGINT	long	oracle.sql.NUMBER
NUMBER	java.sql.Types.REAL	float	oracle.sql.NUMBER
NUMBER	java.sql.Types.FLOAT	double	oracle.sql.NUMBER
NUMBER	java.sql.Types.DOUBLE	double	oracle.sql.NUMBER
RAW	java.sql.Types.BINARY	byte[]	oracle.sql.RAW
RAW	java.sql.Types.VARBINARY	byte[]	oracle.sql.RAW
LONGRAW	java.sql.Types.LONGVARBINARY	byte[]	oracle.sql.RAW
DATE	java.sql.Types.DATE	java.sql.Date	oracle.sql.DATE
DATE	java.sql.Types.TIME	java.sql.Time	oracle.sql.DATE
DATE	java.sql.Types.TIMESTAMP	javal.sql.Timestamp	oracle.sql.DATE
Standard JDBC 2.0 Types			
BLOB	java.sql.Types.BLOB	java.sql.Blob	Oracle.sql.BLOB
CLOB	Java.sql.Types.CLOB	java.sql.Clob	oracle.sql.CLOB

user-defined	java.sql.Types.STRUCT	java.sql.Struct	oracle.sql.STRUCT_object
user-defined	java.sql.Types.REF	java.sql.Ref	oracle.sql.REF_reference
user-defined	java.sql.Types.ARRAY	java.sql.Array	oracle.sql.ARRAY_collection
Oracle Extensions			
BFILE	oracle.jdbc.driver. oracle.sql.BFILE_	n/a	OracleTypes.BFILE
ROWID	oracle.jdbc.driver. oracle.sql.ROWID_	n/a	OracleTypes.ROWID
REFCURSOR type	oracle.jdbc.driver. OracleTypes.CURSOR	java.sql.ResultSet	oracle.jdbc.driver._ OracleResultSet

Exception Handling in JDBC

Like in PL/SQL programs, exceptions do occur in JDBC programs. Notice how the `NO_DATA_FOUND` exception was simulated in the earlier section "Processing the Results of a Database Query That Returns a Single Row."

Exceptions in JDBC are usually of two types:

- Exceptions occurring in the JDBC driver
- Exceptions occurring in the Oracle 8i database itself

Just as PL/SQL provides for an implicit or explicit `RAISE` statement for an exception, Oracle JDBC programs have a `throw` statement that is used to inform that JDBC calls throw the SQL exceptions. This is shown below.

```
throws SQLException
```

This creates instances of the class `java.sql.SQLException` or a subclass of it.

And, like in PL/SQL, SQL exceptions in JDBC have to be handled explicitly. Similar to PL/SQL exception handling sections, Java provides a `try..catch` section that can handle all exceptions including SQL exceptions. Handling an exception can basically include retrieving the error code, error text, the SQL state, and/or printing the error stack trace. The `SQLException` class provides methods for obtaining all of this information in case of error conditions.

Retrieving Error Code, Error Text, and SQL State

There are the methods `getErrorCode()` and `getMessage()` similar to the functions `SQLCODE` and `SQLERRM` in PL/SQL. To retrieve the SQL state, there is the method `getSQLState()`. A brief description of these methods is given below:

- getErrorCode()
- This function returns the five-digit ORA number of the error in case of exceptions occurring in the JDBC driver as well as in the database.
- getMessage()
- This function returns the error message text in case of exceptions occurring in the JDBC driver. For exceptions occurring in the database, this function returns the error message text prefixed with the ORA number.
- getSQLState()

- This function returns the five digit code indicating the SQL state only for exceptions occurring in the database.

The following code illustrates the use of exception handlers in JDBC:

```
try { <JDBC code> }
catch (SQLException e) { System.out.println("ERR: "+ e.getMessage())}
```

We now show the `QueryExample` class of the earlier section with complete exception handlers built in it. The code is as follows:

```
//Import JDBC package
import java.sql.*;
// Import Java package for File I/O
import java.io.*;
public class QueryExample {
 public static void main (String[] args) {
 int ret_code;
 try {
  //Load and register Oracle driver
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
 //Establish a connection
 Connection conn =  DriverManager.getConnection("jdbc:oracle:thin:
 @training:1521:Oracle", "oratest", "oratest");
 //Create a Statement object
 Statement sql_stmt = conn.createStatement();
 //Create a ResultSet object, execute the query and return a
 // resultset
 ResultSet rset = sql_stmt.executeQuery("SELECT empno, ename, sal,
 deptno FROM emp ORDER BY ename");
 //Process the resultset, retrieve data in each row, column by column
 // and write to an operating system file
String str = "";
while (rset.next())
 {
 str += rset.getInt(1)+" "+ rset.getString(2)+" "+rset.getFloat(3)+
 " "+rset.getInt(4)+"\n";
 }
 byte buf[] = str.getBytes();
OutputStream fp = new FileOutputStream("query1.lst");
fp.write(buf);
fp.close();
//Close the ResultSet and Statement
 rset.close();
 sql_stmt.close();
 //Close the database connection
 conn.close();
} catch (SQLException e) {ret_code = e.getErrorCode();
 System.err.println("Oracle Error: "+ ret_code + e.getMessage());}
 catch (IOException e) {System.out.println("Java Error: "+
 e.getMessage()); }
 }
}
```

Printing Error Stack Trace

The SQLException has the method `printStackTrace()` for printing an error stack trace. This method prints the stack trace of the throwable object to the standard error stream.

The following code illustrates this:

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

Share This Save To Your Account

< Back **Page 2** of 7 Next >