Expert Oracle JDBC Programming

R. M. MENON

Apress®

ISBN (pbk): 1-59059-407-X

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Tony Davis
Technical Reviewers: Rob Harrop, Thomas Kyte, Torben Holm, Julian Dyke
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Sofia Marchant
Copy Editor: Nicole LeClerc
Production Manager: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Van Winkle Design Group
Proofreader: Nancy Sixsmith
Indexer: Broccoli Information Management
Artist: Diana Van Winkle, Van Winkle Design Group
Interior Design: Diana Van Winkle, Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 5

Statement and PreparedStatement

In this chapter, you'll briefly look at how Oracle processes SQL statements and then start your journey into the world of statements in JDBC. JDBC statements provide a mechanism for creating and executing SQL statements to query and modify the data stored in a database. As a quick introduction, JDBC offers the following flavors of statement interfaces:

- `Statement`: A `Statement` object lets you execute SQL statements, but does not allow you to vary input variables to the statement at runtime. An example of using this interface appears in Chapter 3.
- `PreparedStatement`: A `PreparedStatement` object represents a precompiled SQL statement that can be executed multiple times. It extends the `Statement` interface and adds the ability to use bind variables. *Bind variables* are parameter markers represented by ? in the SQL string, and they are used to specify input values to the statement that may vary at runtime.
- `CallableStatement`: This interface extends `PreparedStatement` with methods to execute and retrieve results from stored procedures.

You can browse the `javadoc` API for these and other JDBC classes and interfaces at `http://java.sun.com`. In this chapter, we'll focus on the `Statement` and `PreparedStatement` interfaces and their Oracle extensions. We'll cover `CallableStatement` and its Oracle extensions in the next chapter. By the end of this chapter, I hope to convince you that, in production code, you should *always* use `PreparedStatement` (or `CallableStatement`) objects instead of `Statement` objects. In fact, in the next chapter, I make a strong case for almost exclusively using `CallableStatement` in production code.

Before starting the discussion of the `Statement` objects, let's take a quick look at how Oracle processes SQL statements submitted by a client (through SQL*Plus, a JDBC application, etc.). This information will be useful in helping us arrive at certain performance-related conclusions later in this chapter.

Overview of How Oracle Processes SQL Statements (DML)

For this discussion, we only consider Data Manipulation Language (DML) statements. In particular, we exclude Data Definition Language (DDL) statements, as these are typically (and should be) done at install time and are not part of the application code.

DML statements are the statements that you will encounter most often, as you use them to query or manipulate data in existing schema objects. They include `select`, `insert`, `update`, `delete`, and `merge` statements. Oracle goes through the following stages to process a DML statement:

1. *Parsing*: In this step, the statement's syntax and semantics are parsed.
2. *Generating an execution plan*: For each statement, an execution plan is generated and stored in a shared memory area called the *shared pool* (see the section "Memory Structures: Shared Pool" in Chapter 2).
3. *Executing:* The statement executes, using the plan generated in step 2.

Step 2, generating the execution plan, can be very CPU-intensive. To skip this step in most cases, Oracle saves the results of the execution plan in a shared memory structure called the shared pool (see the section "Shared Pool and Bind Variables" of Chapter 2). When you submit a statement to Oracle, as part of the first step of parsing, it checks against the shared pool to see if the same statement was submitted by your session or some other, earlier session. If Oracle does not find the statement in the shared pool, it has to go through all three steps. This phenomenon is called a *hard parse*. On the other hand, if Oracle gets a hit in its shared pool cache, then it can skip the second step of generating the execution plan and directly go to the execution step. This phenomenon is called a *soft parse*.

■ **Note** There is a third category of parsing loosely called ***softer soft parse***. This happens if you have enabled your session to cache a set of cursors related to statements (we look at session cached cursors in more detail in Chapter 14). If there is a hit in a session cache, Oracle does a check in the shared pool to see if the cached cursor points to a valid SQL statement (the statement could become invalid for a variety of reasons, such as schema changes). If the entry is valid, then Oracle can reuse the results of an earlier soft parse done for this statement and go directly to execution step. Basically, this avoids repeated soft parses and saves Oracle resources, thus improving scalability of the applications even further.

Figure 5-1 shows the steps Oracle takes to execute a DML statement.

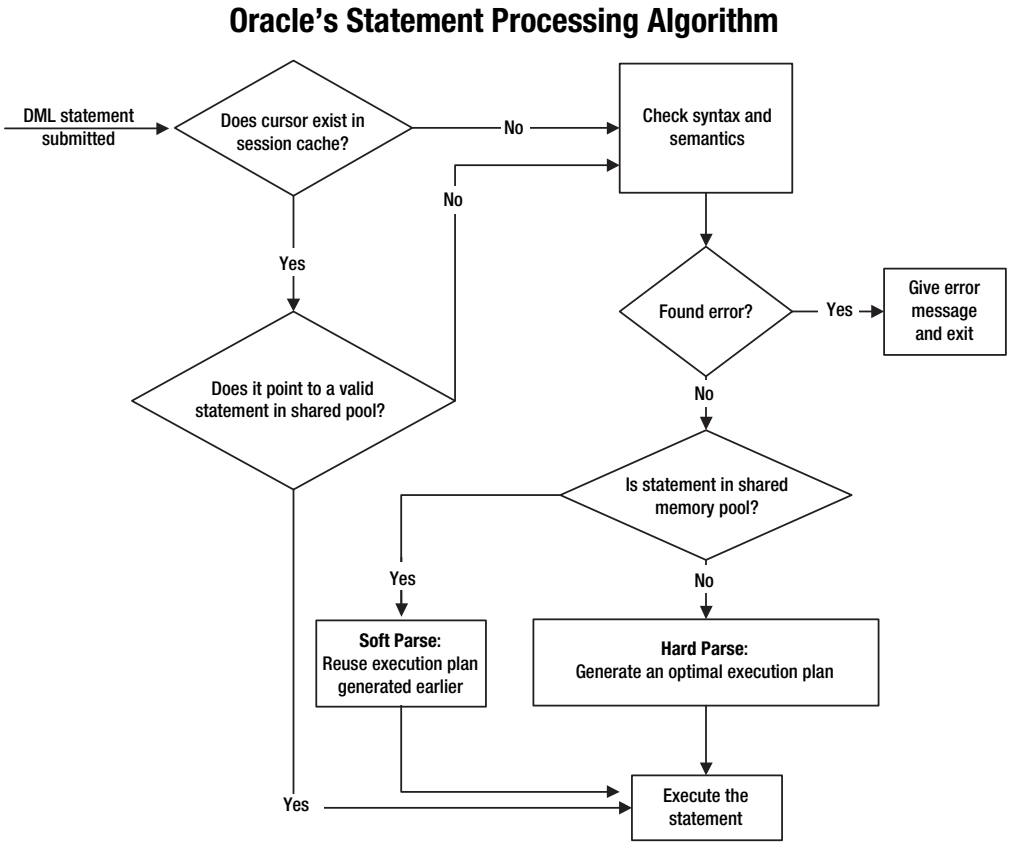

Figure 5-1. *The steps Oracle takes to process and execute a DML statement (see* `http://asktom.oracle.com` *and Chapter 5 of Tom Kyte's* Effective Oracle by Design *[Osborne McGraw-Hill, ISBN: 0-07-223065-7] for a detailed explanation of this algorithm)*

The goal when writing SQL statements in Oracle is to avoid repeated hard parsing and to minimize soft parsing. In this chapter, we'll focus on how to avoid hard parsing in JDBC programs. In Chapter 14, we'll cover techniques for minimizing soft parsing.

JDBC API for Statements

You're now ready to enter the exciting world of statements in JDBC. Recall that the standard JDBC API consists of two packages (see the section "Overview of JDBC API" in Chapter 3):

- `java.sql`: Contains the core JDBC API to access and manipulate information stored in a database, which includes the `Statement` interface and those that inherit from it (e.g., `PreparedStatement` and `CallableStatement`)
- `javax.sql`: Contains APIs for accessing server-side data sources from JDBC clients

Oracle's core JDBC implementation lies in the following two packages:

- `oracle.jdbc` *(and packages beneath it)*: Implements and extends functionality provided by `java.sql` and `javax.sql` interfaces (e.g., `OraclePreparedStatement` and `OracleCallableStatement`)
- `oracle.sql`: Contains classes and interfaces that provide Java mappings to SQL data types (e.g., `oracle.sql.OracleTypes`)

Figure 5-2 shows the JDBC classes pertinent to statements (also shown are the `Connection` and `ResultSet` interfaces, since they are relevant to most JDBC code using statements).

Connection, ResultSet, and Statement Interfaces

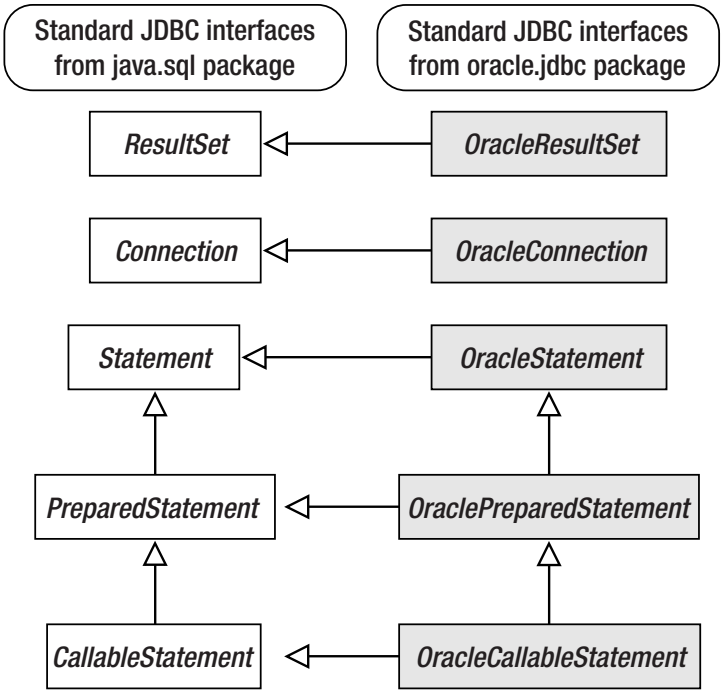

Figure 5-2. *JDBC* `Connection`, `ResultSet`, *and* `Statement` *interfaces and the implementing (or extending) Oracle interfaces*

On the left side of Figure 5-2 are JDBC interfaces in `java.sql` package, and on the right side are the corresponding Oracle interfaces in the `oracle.jdbc` package. Note that `OracleStatement` is an interface that extends `Statement`, `OraclePreparedStatement` is an interface that extends both `OracleStatement` and `PreparedStatement`, and so on.

■ **Tip** I frequently use the command `javap` (available with the JDK) to examine the public methods of a class or an interface. For example, for finding out all public methods of the class `oracle.jdbc.OraclePreparedStatement`, you can execute the following command (after setting the environment and `CLASSPATH` as explained in Chapter 3):

```
javap oracle.jdbc.OraclePreparedStatement
```

In general, the prefix `Oracle` denotes an interface or a class that extends a JDBC standard interface or class, and provides its own Oracle-specific extensions in addition to the standard JDBC API functionality. For example, `java.sql.Connection` is a JDBC standard interface, and `oracle.jdbc.OracleConnection` is an interface that extends `java.sql.Connection`. Table 5-1 shows an overview of Oracle's key interfaces related to `Connection`, `Statement`, and `ResultSet` functionality.

Table 5-1. *JDBC Standard and Oracle Proprietary Interfaces Related to* `Connection`, `Statement`, *and* `ResultSet`

Class or Interface in the `oracle.jdbc` Package	Extends or Implements	Main Functionality
`OracleConnection`	`java.sql.Connection`	Encapsulates a database connection. It has methods to return Oracle statement objects and methods to set Oracle performance extensions for any statement executed by the current connection.
`OracleStatement`	`java.sql.Statement`	Has methods to execute SQL statements (including stored procedures) without bind variables.
`OraclePreparedStatement`	`java.sql.PreparedStatement`, `OracleStatement`	Has methods to execute SQL statements (including stored procedures) with bind variables. In the case of stored procedures, you cannot retrieve any result values back using `PreparedStatement`.
`OracleCallableStatement`	`OraclePreparedStatement`, `java.sql.CallableStatement`	Adds methods to `PreparedStatement` to execute and retrieve data from stored procedures.
`OracleResultSet`	`java.sql.ResultSet`	Contains data representing a data base result set, which is obtained by executing queries against a database.

The Statement Interface

The `Statement` interface is used to execute a SQL statement and return its results to the JDBC program. Chapter 3 presented an example of using this interface in the class `GetEmpDetails`. In this section, we will cover how to query and modify data using the `Statement` interface. For use in our example, we first create a simple table, `t1`, and a PL/SQL procedure, `p2`, that inserts a row in table `t1` as shown:

```
scott@ORA10G> create table t1
  2  (
  3    x number
  4  );
Table created.
scott@ORA10G> create or replace procedure p2( p_x in number )
  2  as
  3  begin
  4    insert into t1 values( p_x );
  5  end;
  6  /
Procedure created.
```

Assuming you have a connection object initialized (as explained in Chapter 3), the steps involved in using a `Statement` interface are as follows:

1. Create a `Statement` object for the SQL statement:

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

2a. The method used to execute a `Statement` object depends on the type of SQL statement being executed. If you want to execute a query using a `select` statement, then use the `executeQuery()` method:

```
public ResultSet executeQuery(String sql) throws SQLException;
```

2b. If you want to execute a data-modifying statement such as `insert`, `delete`, `update`, etc., or a SQL statement that does not return anything, such as a DDL statement, use the `executeUpdate()` method of the `Statement` object. The method returns either the row count for the `insert`, `update`, `delete`, or `merge` statement, or `0` for SQL statements that return nothing. The signature of the method follows:

```
public int executeUpdate(String sql) throws SQLException;
```

2c. If you don't know the statement type, you can use the `execute()` method of the `Statement` interface. For example, if the statement string is a query and you don't know that (because, for example, it is in a variable passed to you by some other program), you could use the `execute()` method:

```
public boolean execute(String sql) throws SQLException;
```

2d. If you want to execute a stored procedure (without using bind variables and without being able to retrieve data returned from the procedure), you can use the `execute()` method.

■ **Caution** As you may have already guessed, using the `Statement` interface for executing stored procedures is ***not*** a good idea. You should use `CallableStatement`, as explained in the next chapter, for this purpose because it allows you to pass parameters as bind variables and it also allows you to retrieve values returned by a stored procedure.

The following `DemoStatement` class illustrates the methods in the `Statement` interface. We first look at the `main()` method after importing the requisite classes:

```
/* This program demonstrates how to use the Statement interface
* to query and modify data and execute stored procedures.
* Note that you should not use the Statement class for executing
* SQL in your production code since it does not allow you to
* use bind variables. You should use either the PreparedStatement
* or the CallableStatement class.
*  COMPATIBLITY NOTE: runs successfully against 9.2.0.1.0 and 10.1.0.2.0
*/
import java.sql.ResultSet;
import java.sql.Date;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Connection;
import book.util.JDBCUtil;
public class DemoStatement
{
  public static void main(String args[])
  {
    Connection conn = null;
    try
    {
      conn = JDBCUtil.getConnection("scott", "tiger", args[0]);
      _demoQuery( conn );
      _demoInsert( conn );
      _demoExecute( conn, "select empno, ename from emp where job = 'CLERK'" );
      _demoExecute( conn, "insert into t1( x) values( 2 ) " );
      _demoInvokingSQLProcedure( conn );
      conn.commit();
    }
    catch (SQLException e)
    {
      // handle the exception - in this case, we
      // roll back the transaction and
      // print an error message and stack trace.
      JDBCUtil.printExceptionAndRollback ( conn, e );
    }
    finally
```

```
    {
      // release resources associated with JDBC
      // in the finally clause.
      JDBCUtil.close( conn );
    }
  } // end of main
```

In the `main()` method, we get the connection as `SCOTT`, using the `JDBCUtil.getConnection()` method as explained in Chapter 3. We invoke the following methods, which I explain shortly:

- `_demoQuery`: Demonstrates executing a query using the `Statement` interface
- `_demoInsert`: Demonstrates executing an insert using the `Statement` interface
- `_demoExecute`: Demonstrates executing any DML (a query or an insert, update, etc.) using the `Statement` interface
- `_demoInvokingSQLProcedure`: Demonstrates invoking a SQL procedure without using bind variables and without being able to retrieve values back from the stored procedure

Let's look at each of these methods in detail now, starting with the first half of `_demoQuery()`:

```
// execute a query using the Statement interface
private static void _demoQuery( Connection conn ) throws SQLException
{
  ResultSet rset = null;
  Statement stmt = null;
  try
  {
```

Inside the `try catch` block, we first create the statement:

```
    stmt = conn.createStatement();
```

Next, we use the `executeQuery()` method on the `Statement` object, passing the `select` statement that we want to execute. The invocation returns query results in the form of a `ResultSet` object.

```
    // execute the query
    rset = stmt.executeQuery(
      "select empno, ename, hiredate from emp where job = 'CLERK'" );
```

As explained in Chapter 3, a `ResultSet` object maintains a cursor pointing to its current row of data. Initially, the cursor is positioned before the first row. We use the `next()` method to move the cursor to the next row, thus iterating through the result set as shown in the following code. The `next()` method returns `false` when there are no more rows in the `ResultSet` object, at which point we exit the loop. Within the loop, we retrieve each column value of a row using the appropriate `getXXX()` method of the `ResultSet` interface. This means using `getInt()` with the integer column `empno`, `getString()` with the string column `ename`, and `getDate()` with the

date column `hiredate`. The first parameter of these methods is the positional index of the column in the `select` clause in the query (the index starts from 1).

```
    // loop through the result set and print
    while (rset.next())
    {
      int empNo = rset.getInt ( 1 );
      String empName = rset.getString ( 2 );
      Date hireDate = rset.getDate ( 3 );
      System.out.println( empNo + "," + empName + "," + hireDate );
    }
```

We end the `try catch` block with a `finally` clause in which we close the result set and the statement objects. Putting these objects in the `finally` clause ensures that they always get called (e.g., even in the case of an exception); otherwise, the database can run out of cursor resources.

```
    }
    finally
    {
      JDBCUtil.close( rset );
      JDBCUtil.close( stmt );
    }
  }
```

The next method, `_demoInsert()`, illustrates how to insert data using the `executeUpdate()` method of the `Statement` interface:

```
  // demonstrate inserting record using the Statement interface
  private static void _demoInsert( Connection conn ) throws SQLException
  {
    Statement stmt = null;
    try
    {
      stmt = conn.createStatement();
      // execute the insert
      int numOfRowsInserted = stmt.executeUpdate(
        "insert into t1( x) values( 1 ) " );
      System.out.println( "Number of rows inserted = " + numOfRowsInserted );
    }
    finally
    {
      JDBCUtil.close( stmt );
    }
  }
```

As you can see, most of the code is similar to the method `_demoQuery()` shown earlier. The only difference is that this time we use the `executeUpdate()` method to insert a row in the table `t1` and print the number of rows inserted successfully as returned by `executeUpdate()`. The same technique can also be used to update and delete rows from a table.

The following method, `_demoExecute()`, takes a connection object and a SQL statement and executes the statement using the `execute()` method of the `Statement` interface. It can be invoked for a query statement as well as a nonquery DML statement, such as an insert, as illustrated in the `main()` program:

```
// demonstrate the execute() method of the Statement interface
private static void _demoExecute( Connection conn, String sqlStmt )
  throws SQLException
{
  ResultSet rset = null;
  Statement stmt = null;
  try
  {
    stmt = conn.createStatement();
    // execute the query
```

After creating the statement, we execute it and get the `boolean` value that tells us if the statement was a query or a not.

```
    boolean isQuery = stmt.execute( sqlStmt );
```

If it is a query, we get the `ResultSet` and print the results. In this example, we have to know the column type and position (or name) of the query to retrieve the results. Notice how we use the column names (instead of column's positional index) to get the results this time.

```
   // if it is a query, get the result set and print the results
    if( isQuery )
    {
      rset = stmt.getResultSet();
      while (rset.next())
      {
        int empNo = rset.getInt ( "empno" );
        String empName = rset.getString ( "ename" );
        System.out.println( empNo + "," + empName );
      }
    }
```

If it is not a query, we assume it is an `insert`, `update`, or `delete` and get the number of rows affected using the `getUpdateCount()` method. We also close the statement and result set at the end of the method:

```
    else
    {
      // we assume it is an insert, update, or delete statement
      int numOfRowsAffected = stmt.getUpdateCount();
      System.out.println( "Number of rows affected by execute() = " +
        numOfRowsAffected );
    }
  }
  finally
```

```
    {
      JDBCUtil.close( rset );
      JDBCUtil.close( stmt );
    }
  }
```

As mentioned earlier, we can also use the `execute()` method to execute a stored procedure, although we cannot use bind values, and we cannot retrieve any values returned by the stored procedure. The method `_demoInvokingSQLProcedure` at the end of the program `DemoStatement` illustrates this:

```
  private static void _demoInvokingSQLProcedure( Connection conn )
    throws SQLException
  {
    Statement stmt = null;
    try
    {
      stmt = conn.createStatement();
      // execute the sql procedure
      boolean ignore = stmt.execute( "begin p2( 3 ); end;" );
    }
    finally
    {
      JDBCUtil.close( stmt );
    }
  }
} // end of program
```

We use the Oracle style anonymous block enclosed in the `begin/end` block to invoke the procedure. We will look at another style, called SQL92, in Chapter 6.

Now that you've learned about the `Statement` interface, let's look at why it isn't a good idea to use it in your code. The basic problem with the `Statement` interface is that it accepts only literal parameters. In the preceding example, the value of employee number would be hard-coded into the query each time, and every variation of the query—even though it may vary only by the value of supplied employee number—would be treated as a brand-new query by Oracle and would be hard-parsed. For example, the query `select ename from emp where empno = 1234` is treated as distinct from the query `select ename from emp where empno = 4321`, whereas both queries are the same except for the user input of the employee number. As already discussed, this is something to avoid. When we use a `PreparedStatement` object, we replace the literal value with a placeholder (`?` in JDBC)—in other words, a *bind variable*—so that Oracle will treat it as the same statement each time. The actual values (1234 and 4321 in the example) are bound to the query at runtime. In this case, Oracle performs a hard parse the first time it encounters the statement, and it performs much less expensive soft parses subsequently.

If you want to see proof of how much difference this can make in terms of performance, I refer you to the example in the "Bind Variables Example" section of the Chapter 2, which showed the vast decrease in execution time and resource uses obtained when inserting 10,000 records in a table with bind variables rather than without.

Another problem with using the `Statement` interface is that the program becomes vulnerable to SQL injection attacks by hackers. We'll look at SQL injection in more detail when we cover the `PreparedStatement` interface in the upcoming section titled "Using Bind Variables Makes Your Program More Secure."

■ **Tip** Never use the `Statement` class in your production code, as it does not allow you to use bind variables, which in turn makes your program slower, less scalable, and more vulnerable to SQL injection attacks. To use bind variables in JDBC, you have to use `PreparedStatement/OraclePreparedStatement` (or `CallableStatement` or `OracleCallableStatement` in the case of stored procedures, as we will see in the next chapter) instead of the `Statement` class.

The PreparedStatement Interface

A `PreparedStatement` object represents a precompiled SQL statement that lets you efficiently execute a statement multiple times using different bind variables. Using prepared statements lets Oracle compile the statement only once instead of having to compile it once with each call that changes one of the input parameters. An analogy would be a Java program that takes some input parameters.

The first option is to hard-code your input parameter values in the program. In this case, every time you need to deal with a different input value, you will have to change the program and recompile it. Using a `Statement` class is somewhat similar to this: Oracle has to compile your statement each time an input value changes, since the value is hard-coded in the statement itself.

A second, smarter option is to get the user input as a command-line parameter. In this case, you compile the program only once before invoking it many times with different values of the command-line parameters. Using `PreparedStatement` with bind values is similar to this scenario, in that you compile the statement only once and bind it at runtime with different values.

The next few sections discuss the `PreparedStatement` interface in detail.

Creating a PreparedStatement Object

The first step is to create a `PreparedStatement` object by invoking the `prepareStatement()` method of the `Connection` object, whose signature follows:

```
public PreparedStatement prepareStatement( String sql) throws SQLException
```

This method takes a statement and compiles it. Later, we can execute the same statement binding it with different values at runtime, for example:

```
PreparedStatement pstmt = conn.prepareStatement(
"select empno, ename, job from emp where empno = ? and hiredate < ?");
```

Notice how the actual values of the employee number and the hire date have been replaced by the literal ?. The ? in the query string acts as the placeholder for input variables that need to be bound subsequently in a statement. Let's now look at how we can use bind variables when working with a `PreparedStatement` object.

Using Bind Variables with PreparedStatements

There are two ways of binding parameters to the SQL statement:

- *By parameter index or ordinal position*: In this case, you use the parameter's index position to bind the parameter. The indexes begin with 1.
- *By parameter name*: In this case, you bind the parameter by its name. This requires the use of Oracle extension methods in the `OraclePreparedStatement` interface.

Binding Parameters by Index (or by Ordinal Position)

To bind a parameter by index, we use the appropriate `setXXX()` method depending on the data type of the input variable being bound. Here the index refers to the ordinal position of the ? value in the query string. For example, consider the SQL statement

```
select empno, ename, job from emp where empno = ? and hiredate < ?"
```

In the preceding statement, the first literal value has to be replaced by an integer representing the employee number, so we will use the `setInt()` method of the `PreparedStatement` interface:

```
public void setInt(int parameterIndex, int x) throws SQLException;
```

Similarly, the second literal value is a date, so we use the `setDate()` method to bind it with a date value:

```
public void setDate(int parameterIndex, java.sql.Date date ) throws SQLException
```

Next, we'll look at how to bind parameters by name.

Binding Parameters by Name (Oracle 10*g* Only)

An alternative to using ? as a placeholder for our bind variables is to bind by parameter name. This is an Oracle 10*g*-specific feature that improves the readability of the prepared statement string.

To use named parameters, we have to use the appropriate `setXXXAtName()` method of the `OraclePreparedStatement` interface. For example, if we want to bind the query discussed in the previous section by name, we would first use the following query string while preparing the statement:

```
select empno, ename, job from emp where empno = :empno and hiredate < :hiredate"
```

Notice that the literal placeholder ? has been replaced by a parameter name of our choice preceded with a colon (:). We then use `setIntAtName()` of the `OraclePreparedStatement` interface for the first parameter and `setDataAtName()` for the second parameter:

```
public void setIntAtName(java.lang.String parameterName,
  java.sql.Date value) throws SQLException;
public void setDateAtName(java.lang.String parameterName,
  java.sql.Date value) throws SQLException;
```

Executing a PreparedStatement

To execute a `PreparedStatement`, you can use one of the following three methods:

```
public boolean execute()throws SQLException
public ResultSet executeQuery()throws SQLException
public int executeUpdate()throws SQLException
```

The logic of when to use each method is the same as that for the methods with the same names in the `Statement` interface discussed in the section "The Statement Interface." Notice, however, that unlike their counterparts in `Statement` interface, these methods don't take a SQL string. This is because the SQL statement itself has already been precompiled at the time you invoke the `prepareStatement()` method of the `Connection` object.

It's time for some examples that illustrate all of the steps just described. Let's first look at an example that queries data.

Example of Using PreparedStatement to Query Data

The class `DemoPreparedStatementQuery` described in this section illustrates how to use the `PreparedStatement` interface in JDBC programs to select data from a database. It illustrates binding by parameter index and binding by parameter name. After the necessary imports, we have the `main()` method of the class:

```
/* This program demonstrates how to query data from a table
* using the PreparedStatement interface. It illustrates
* binding a parameter both by index and by name.
* COMPATIBLITY NOTE: runs successfully against 10.1.0.2.0.
*    against 9.2.0.1.0, you have to comment out the
*    code using the binding by name feature to compile and
*    run this, as bind by name is not supported in 9i.
*/
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.Connection;
import oracle.jdbc.OraclePreparedStatement;
import book.util.JDBCUtil;
import book.ch03.Util;
class DemoPreparedStatementQuery
{
```

```
  public static void main(String args[])
  {
    Util.checkProgramUsage( args );
    Connection conn = null;
    try
    {
      conn = JDBCUtil.getConnection("scott", "tiger", args[0]);
      _demoBindingByParameterIndex( conn );
      _demoBindingByParameterName( conn );
    }
    catch (SQLException e)
    {
      // handle the exception properly - in this case, we just
      // print the stack trace.
      JDBCUtil.printException ( e );
    }
    finally
    {
      // release the JDBC resources in the finally clause.
      JDBCUtil.close( conn );
    }
  } // end of main()
```

In the `main()` method, after getting the JDBC connection, we invoke two methods:

- `demoBindingByParameterIndex()`: Demonstrates binding by parameter index
- `demoBindingByParameterName()`: Demonstrates binding by parameter name

We then close the connection in the `finally` clause to end the `main()` method.

The method `_demoBindingByParameterIndex()` starts by declaring required variables and beginning a `try catch` block (notice the constants declared for column indexes later):

```
/* demo parameter binding by index */
  private static void _demoBindingByParameterIndex( Connection conn )
    throws SQLException
  {
    String stmtString =
      "select empno, ename, job from emp where job = ? and hiredate < ?";
    System.out.println( "\nCase 1: bind parameter by index");
    System.out.println( "Statement: " + stmtString );
    PreparedStatement pstmt = null;
    ResultSet rset = null;
    final int JOB_COLUMN_INDEX = 1;
    final int HIREDATE_COLUMN_INDEX = 2;
    final int SELECT_CLAUSE_EMPNO_COLUMN_INDEX = 1;
    final int SELECT_CLAUSE_ENAME_COLUMN_INDEX = 2;
    final int SELECT_CLAUSE_JOB_COLUMN_INDEX = 3;
    try
    {
```

Notice how the select statement has ? for input parameters. The query will get us all employees of a given job title and hire date earlier than a given date. Next, we prepare the statement

```
pstmt = conn.prepareStatement( stmtString );
```

We then bind the parameters. The first parameter is a string for the `job` column of the `emp` table; hence we use the `setString()` method, passing the constant that defines the `job` column index value of 1 and the parameter value of `CLERK`.

```
pstmt.setString(JOB_COLUMN_INDEX, "CLERK" );
```

For the `hiredate` column, we pass the current date. The parameter index is the constant `HIREDATE_COLUMN_INDEX` with the value `2` in this case:

```
pstmt.setDate(HIREDATE_COLUMN_INDEX, new java.sql.Date(
  new java.util.Date().getTime()));
```

Notice that the date value is of type `java.sql.Date`, not `java.util.Date`.

We execute the statement next. Since it is a query, we use the `executeQuery()` method:

```
rset = pstmt.executeQuery();
```

Finally, we end the method after printing the results of the query and closing the result set and statement:

```
    // print the result
    System.out.println( "printing query results ...\n");
    while (rset.next())
    {
      int empNo = rset.getInt ( 1 );
      String empName = rset.getString ( 2 );
      String empJob = rset.getString ( 3 );
      System.out.println( empNo + " " + empName + " " + empJob );
    }
  }
  finally
  {
    // release JDBC-related resources in the finally clause.
    JDBCUtil.close( rset );
    JDBCUtil.close( pstmt );
  }
}
```

Let's look at how we can execute the same query, but this time binding parameters by name. The method `_demoBindingByParameterName()` begins by declaring variables and starting a `try catch` block:

```
private static void _demoBindingByParameterName( Connection conn )
  throws SQLException
{
  String stmtString = "select empno, ename, job " +
```

```
      "from emp where job = :job and hiredate < :hiredate";
    System.out.println( "\nCase 2: bind parameter by name\n");
    System.out.println( "Statement: " + stmtString );
    OraclePreparedStatement opstmt = null;
    ResultSet rset = null;
    final int SELECT_CLAUSE_EMPNO_COLUMN_INDEX = 1;
    final int SELECT_CLAUSE_ENAME_COLUMN_INDEX = 2;
    final int SELECT_CLAUSE_JOB_COLUMN_INDEX = 3;
    try
    {
```

Note that this time we use the parameter names `:job` and `:hiredate` for our input parameters. Notice also that we have to use the `OraclePreparedStatement` interface. The first step involves preparing the statement with the query:

```
      opstmt = (OraclePreparedStatement) conn.prepareStatement( stmtString );
```

Next, we bind the `job` parameter with the value `CLERK` using the `setStringAtName()` method of the `OraclePreparedStatement` interface (note that there is no `:` in the string we pass as the parameter name):

```
opstmt.setStringAtName("job", "CLERK" );
```

We bind the `hiredate` parameter with the current date value:

```
     opstmt.setDateAtName("hiredate", new java.sql.Date(
       new java.util.Date().getTime()));
```

The next steps of executing the query, printing the results, and releasing the resources are the same as in the previous example. This also ends our class listing.

```
      // execute the query
      rset = opstmt.executeQuery();
      // print the result
      System.out.println( "printing query results ...\n");
      while (rset.next())
      {
        int empNo = rset.getInt ( SELECT_CLAUSE_EMPNO_COLUMN_INDEX );
        String empName = rset.getString ( SELECT_CLAUSE_ENAME_COLUMN_INDEX );
        String empJob = rset.getString ( SELECT_CLAUSE_JOB_COLUMN_INDEX );
        System.out.println( empNo + " " + empName + " " + empJob );
      }
    }
    finally
    {
      // release JDBC-related resources in the finally clause.
      JDBCUtil.close( rset );
      JDBCUtil.close( opstmt );
    }
  }
} // end of program
```

This is the sample execution output of the `DemoPreparedStatementQuery` program:

```
B:\code\book\ch05>java DemoPreparedStatementQuery ora10g
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(PORT=1521)(HOST=rmeno
n-lap))(CONNECT_DATA=(SID=ora10g)))

Case 1: bind parameter by index
Statement: select empno, ename, job from emp where job = ? and hiredate < ?
printing query results ...

7369 SMITH CLERK
7876 ADAMS CLERK
7900 JAMES CLERK
7934 MILLER CLERK

Case 2: bind parameter by name

Statement: select empno, ename, job from emp where job = :job and hiredate < :hir
edate
printing query results ...

7369 SMITH CLERK
7876 ADAMS CLERK
7900 JAMES CLERK
7934 MILLER CLERK
```

So should you bind parameters by index or by name? This choice comes into play only if you are using Oracle 10*g*. In my benchmark tests, I found no material difference in performance between binding by index and binding by name. If portability across databases is critical for you, you should bind parameters by index. Otherwise, using `OraclePreparedStatement` and binding by parameter name can marginally improve the readability of your code. You can (and should) also improve readability in the case of binding by parameter index by defining meaningful constants for the parameter indexes as we did here (in other examples in this book, we may not follow this convention for simplicity). However, the SQL constants for statement strings still contain the not-so-readable ? in this case.

■ **Caution** Under certain circumstances, previous versions of the Oracle JDBC drivers allowed binding `PreparedStatement` variables by name when using the standard `setXXX` methods. This capability to bind by name using the `setXXX` methods is ***not*** part of the JDBC specification, and Oracle ***does not*** support it. The JDBC drivers can throw a `SQLException` or produce unexpected results if you use this method, so I ***strongly*** recommend that you not use this technique.

Example of Using PreparedStatement to Modify Data

In this section, we'll look at how to make some modifications to existing data in our database. First, let's create a table, `t1`, and insert some data in it in the `BENCHMARK` schema as follows:

```
benchmark@ORA10G> create table t1 ( x number primary key,
  2  y varchar2(100),
  3  z date );
Table created.
benchmark@ORA10G> insert into t1 values ( 1, 'string 1', sysdate+1 );
1 row created.
benchmark@ORA10G> insert into t1 values ( 2, 'string 2', sysdate+2 );
1 row created.
benchmark@ORA10G> insert into t1 values ( 3, 'string 3', sysdate+3 );
1 row created.
benchmark@ORA10G> insert into t1 values ( 4, 'string 4', sysdate+4 );
1 row created.
benchmark@ORA10G> commit;
```

The following `DemoInsUpdDelUsingPreparedStatement` class illustrates how to use a prepared statement to insert, update, and delete data from table `t1`. The program begins by importing statements and defining the `main()` method that invokes other private methods:

```
/* This program shows how to insert, update, and delete data using
   the PreparedStatement interface.
* COMPATIBLITY NOTE: runs successfully against 10.1.0.2.0.
*    against 9.2.0.1.0, you have to comment out the
*    code using the binding by name feature to compile and
*    run this, as bind by name is not supported in 9i.
*/
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.Connection;
import oracle.jdbc.OraclePreparedStatement;
import book.util.JDBCUtil;
import book.util.Util;
class DemoInsUpdDelUsingPreparedStatement
{
  public static void main(String args[])
  {
    Util.checkProgramUsage( args );
    Connection conn = null;
    PreparedStatement pstmt = null;
    try
    {
      // get connection
      conn = JDBCUtil.getConnection("benchmark", "benchmark", args[0]);
      _demoInsert( conn );
      _demoUpdate( conn );
```

```
      _demoDelete( conn );
      conn.commit();
    }
    catch (SQLException e)
    {
      // handle the exception properly - in this case, we just
      // print a message and roll back
      JDBCUtil.printExceptionAndRollback( conn, e );
    }
    finally
    {
      // release JDBC resources in the finally clause.
      JDBCUtil.close( conn );
    }
  }
```

After getting the connection, the `main()` method invokes three private methods:

- `demoInsert()`: Demonstrates inserting data
- `demoUpdate()`: Demonstrates updating data, and binds parameters by name
- `demoDelete()`: Demonstrates deleting data

The method `_demoInsert()` begins by preparing a statement to insert a row into `t1`:

```
// demo insert
private static void _demoInsert( Connection conn ) throws SQLException
{
  PreparedStatement pstmt = null;
  try
  {
    // prepare the statement
    pstmt = conn.prepareStatement( "insert into t1 values ( ?, ?, ? )");
```

Next, we bind the values for the three columns `x`, `y`, and `z`:

```
    pstmt.setInt(1, 5 ); // bind the value 5 to the first placeholder
    pstmt.setString(2, "string 5" );
    pstmt.setDate(3, new java.sql.Date( new java.util.Date().getTime()));
```

We execute the statement using the `executeUpdate()` method, which returns the number of rows inserted. We print out the number of rows inserted and close the prepared statement to end the method:

```
    int numOfRowsInserted  = pstmt.executeUpdate();
    System.out.println( "Inserted " + numOfRowsInserted + " row(s)" );
  }
  finally
  {
    // release JDBC related resources in the finally clause.
```

```
      JDBCUtil.close( pstmt );
    }
  }
```

The _demoUpdate() method updates one row of table t1. We use binding by parameter name this time. The method begins by creating a prepared statement and casting it to the OraclePreparedStatement interface:

```
  // demo update use bind by name
  private static void _demoUpdate( Connection conn ) throws SQLException
  {
    OraclePreparedStatement opstmt = null;
    try
    {
      // prepare the statement
      opstmt = (OraclePreparedStatement)
        conn.prepareStatement( "update t1 set y = :y where x = :x");
```

We bind the two named parameters x and y next:

```
      // bind the values by name.
      opstmt.setStringAtName("y", "string 1 updated" );
      opstmt.setIntAtName("x", 1 );
```

The process of executing the statement is the same as that in the case of _demoInsert():

```
      // execute the statement
      int numOfRowsUpdated  = opstmt.executeUpdate();
      System.out.println( "Updated " + numOfRowsUpdated + " row(s)" );
    }
    finally
    {
      // release JDBC-related resources in the finally clause.
      JDBCUtil.close( opstmt );
    }
  }
```

We end the program with the _demoDelete() method, which is similar to the _demoInsert() method:

```
  // demo delete
  private static void _demoDelete( Connection conn ) throws SQLException
  {
    PreparedStatement pstmt = null;
    try
    {
      // prepare the statement
      pstmt = conn.prepareStatement( "delete from t1 where x = ?");
      // bind the values
      pstmt.setInt(1, 2 );
```

```
      // execute the statement
      int numOfRowsDeleted  = pstmt.executeUpdate();
      System.out.println( "Deleted " + numOfRowsDeleted + " row(s)" );
    }
    finally
    {
      // release JDBC-related resources in the finally clause.
      JDBCUtil.close( pstmt );
    }
  }
} // end of program
```

When we execute the program `DemoInsUpdDelUsingPreparedStatement`, we get the following output:

```
B:\code\book\ch05>java DemoInsUpdDelUsingPreparedStatement ora10g
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(PORT=1521)(HOST=rmeno
n-lap))(CONNECT_DATA=(SID=ora10g)))
Inserted 1 row(s)
Updated 1 row(s)
Deleted 1 row(s)
```

Using Bind Variables Makes Your Program More Secure

By now you should be convinced that using bind variables is critical for your program's performance and scalability. However, there is another equally important (in some cases, more important) reason to use bind variables. Using bind variables can protect your application from SQL injection attacks.

SQL injection is a technique that enables a hacker to execute unauthorized SQL statements by taking advantage of applications that use input criteria to dynamically build a SQL statement string and execute it. Let's look at an example. Consider an application that stores its username and password information in a table. For simplicity, we'll store this information in an unencrypted form, although in real-world applications better alternatives exist. We first create a table, `user_info`, with two columns, `username` and `password`:

```
benchmark@ORA10G> create table user_info
  2  (
  3    username varchar2(15) not null,
  4    password varchar2(15 ) not null
  5  );
```

We insert ten usernames and passwords. For illustration purposes, the data is such that the user `user1` has the password `password1`, the user `user2` has the password `password2`, and so on.

```
benchmark@ORA10G> begin
  2    for i in 1..10
  3    loop
```

```
  4    insert into user_info( username, password )
  5    values( 'user'||i, 'password'||i );
  6    end loop;
  7  end;
  8  /
PL/SQL procedure successfully completed.
benchmark@ORA10G> select username, password from user_info;
USERNAME        PASSWORD
--------------- ---------------
user1           password1
user2           password2
user3           password3
user4           password4
user5           password5
user6           password6
user7           password7
user8           password8
user9           password9
user10          password10
```

Let's now look at `DemoSQLInjection`, a program that authenticates an application user by validating the combination of username and password input from the command line against table `user_info`'s data. The usage of the program is

```
java DemoSQLInjection <bind|nobind> <username> <password>
```

The program takes three parameters from the command line. The first parameter can have two possible values: `bind` or `nobind`. If we give an option of `nobind`, the program verifies the username and password without using bind variables; otherwise, it does so using bind variables. The second parameter is the username, and the third parameter is the password. The class listing begins with import statements and declaration of the `main()` method:

```
/* This program demonstrates how using bind variables can prevent SQL
     injection attacks.
* COMPATIBLITY NOTE: runs successfully against 9.2.0.1.0 and 10.1.0.2.0.
*/
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.Connection;
import book.util.JDBCUtil;
class DemoSQLInjection
{
  public static void main(String args[])
  {
```

Inside `main()`, we invoke `_validateProgramInputs()` (defined later), which performs simple input validation and prints program usage if required. We then store the three command-line parameters in string variables:

```
_validateProgramInputs( args );
String selectedOption = args[0];
String username = args[1];
String password = args[2];
```

The next step is to get the database connection within the `try catch` block.

```
Connection conn = null;
try
{
  // get connection
  conn = JDBCUtil.getConnection("benchmark", "benchmark", "ora10g");
```

If the first parameter is `nobind`, we invoke the method `_authenticateWithoutUsingBind➥Values()`, which performs the authentication without using bind variables. Otherwise, it invokes `_authenticateUsingBindValues()`, which validates the username and password using bind variables. We end the `main()` method with the usual `catch` and `finally` clauses:

```
    if( NO_BIND.equals( selectedOption ) )
    {
      _authenticateWithoutUsingBindValues( conn, selectedOption,
        username, password );
    }
    else
    {
      _authenticateUsingBindValues( conn, selectedOption, username, password );
    }
  }
  catch (SQLException e)
  {
    // handle the exception properly - in this case, we just
    // print a message and roll back
    JDBCUtil.printExceptionAndRollback( conn, e );
  }
  finally
  {
    // release JDBC resources in the finally clause.
    JDBCUtil.close( conn );
  }
}
```

The definition of `_authenticateWithoutUsingBindValues()` follows. The main point to note is that the query statement string is computed by concatenating the input username and password to the query string. We use the `Statement` class to emphasize that we are not using bind variables in this case.

```
// authenticate without using bind values
private static void _authenticateWithoutUsingBindValues( Connection conn,
  String selectedOption, String username, String password ) throws SQLException
{
  Statement stmt = null;
  ResultSet rset = null;
  try
  {
    stmt = conn.createStatement();
    String verifyStmtString = "select count(*) from user_info " +
                              "where username = '" + username + "'" +
                              " and password = '" + password + "'";
    System.out.println("verify statement: " + verifyStmtString );
```

We execute the query next. If we find no records matching the input username and password, we print a message indicating that the authentication failed. Otherwise, authentication succeeds and a message to that effect is printed:

```
    rset = stmt.executeQuery( verifyStmtString );
    while( rset.next() )
    {
      int count = rset.getInt(1);
      if( count == 0 )
        System.out.println("Invalid username and password - access denied!");
      else
        System.out.println("Congratulations! You have been " +
          "authenticated successfully!");
    }
  }
  finally
  {
    // release JDBC-related resources in the finally clause.
    JDBCUtil.close( rset );
    JDBCUtil.close( stmt );
  }
}
```

The following method, `authenticateUsingBindValues()`, also executes the same `select` statement, except this time we use a `PreparedStatement` object and bind our input parameter values:

```
private static void _authenticateUsingBindValues( Connection conn,
  String selectedOption, String username, String password ) throws SQLException
{
  PreparedStatement pstmt = null;
  ResultSet rset = null;
  try
  {
```

```
    String verifyStmtString = "select count(*) from user_info " +
                              "where username = ? "+
                              " and password = ?";
    System.out.println("verify statement: " + verifyStmtString );
    // prepare the statement
    pstmt = conn.prepareStatement( verifyStmtString );
    // bind the values
    pstmt.setString(1, username );
    pstmt.setString(2, password );
    // execute the statement
    rset = pstmt.executeQuery();
    while( rset.next() )
    {
      int count = rset.getInt(1);
      if( count == 0 )
        System.out.println("Invalid username and password - access denied!");
      else
        System.out.println("Congratulations! You have been " +
          "authenticated successfully!");
    }
  }
  finally
  {
    // release JDBC related resources in the finally clause.
    JDBCUtil.close( rset );
    JDBCUtil.close( pstmt );
  }
}
```

The program ends after defining the `_validateProgramInputs()` method:

```
// check command-line parameters.
private static void _validateProgramInputs( String[] args )
{
  if( args.length != 3 )
  {
    System.out.println(" Usage: java <program_name> " +
      "<bind|nobind> <username> <password>");
    System.exit(1);
  }
  if( !( NO_BIND.equals( args[0] )  || BIND.equals( args[0] ) ) )
  {
    System.out.println(" Usage: java <program_name> " +
      "<bind|nobind> <username> <password>");
    System.exit(1);
  }
}
```

```
  private static final String NO_BIND= "nobind";
  private static final String BIND= "bind";
} // end of program
```

When we execute the preceding program with the `nobind` option while giving a valid username and password, it works fine:

```
B:\code\book\ch05>java DemoSQLInjection  nobind user1 password1
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(PORT=1521)(HOST=rmeno
n-lap))(CONNECT_DATA=(SID=ora10g)))
verify statement: select count(*) from user_info where username = 'user1' and pa
ssword = 'password1'
Congratulations! You have been authenticated successfully!
```

If we use the same option, but give a wrong username password combination, we are denied access, as expected:

```
B:\>java DemoSQLInjection  nobind user1 password2
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(PORT=1521)(HOST=rmenon-lap))(CONNECT_DATA=(SID=ora10g)))
verify statement: select count(*) from user_info where username = 'user1' and pa
ssword = 'password2'
Invalid username and password - access denied!!
```

So far, the program looks rock-solid even if we don't use bind variables. Unfortunately, that is not really the case. Consider the following invocation with the option of `nobind`:

```
B:\> java DemoSQLInjection nobind invalid_user "junk_password' or 'x'='x"
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(PORT=1521)(HOST=rmenon-lap))(CONNECT_DATA=(SID=ora10g)))
verify statement: select count(*) from user_info where username = 'invalid_user'
 and password = 'junk_password' or 'x'='x'
Congratulations! You have been authenticated successfully!
```

Even though an invalid username and password was given, the authentication was successful. What happened? A careful examination reveals that the input was engineered in such a way that the `where` clause of the query had the criterion `" or 'x' = 'x'"` appended to the end. And since this last criterion is always true, the executing `select` statement will always return a nonzero count, resulting in a successful authentication.

Let's see what happens if we use the same input parameters, but choose the `bind` option this time:

```
B:\code\book\ch05>java DemoSQLInjection bind invalid_user "junk_password' or 'x'
='x"
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(PORT=1521)(HOST=localhost))(CONNECT_DATA=(SID=ora10g)))
verify statement: select count(*) from user_info
where username = ?  and password = ?
Invalid username and password - access denied!
```

The hacker would be disappointed in this case. When we use bind variables, the query itself remains the same, since we use ? in place of actual parameter values. Hence, it does not matter what the input parameter values are—the program will work as expected.

The SQL injection attack has caused a lot of grief at many websites that use relational databases. Note that the SQL injection attack is not specific to the Oracle database. Much has been written on this topic, as a simple search on Google will reveal. In Oracle, using bind variables can protect applications from this dangerous attack most of the time.

Now that you have seen how to use bind variables in the `PreparedStatement` and all the benefits of using bind variables, let's move on to look at some nuances related to bind variable usage.

Nuances of Bind Variable Usage

As you know by now, a bind variable is a parameter used in a SQL statement, the value of which is bound at runtime. So, for example, you could have the following statement, in which the values to be inserted into the `emp` table are bound at runtime:

```
PreparedStatement pstmt = conn.prepareStatement(
        "insert into emp values ( ?, ?, ? )");
```

However, can you run the following statement, in which the *table name* is bound at runtime?

```
PreparedStatement pstmt = conn.prepareStatement(
        "insert into ? values ( ?, ?, ? )");
```

The answer is no. If you try to run such code, you will get the exception `java.sql.SQLException: ORA-00903: invalid table name`.

Recall that the concept of bind variables exists so that Oracle can reuse the generated execution plans for a statement by substituting placeholders with literal values. Also, the parsing and query plan generation of a statement occur *before* the bind variables are evaluated. In the preceding case, for example, the parsing cannot be done because the optimizer needs to know the table name to generate a plan, to carry out the semantic checks (e.g., whether the user has the privilege to insert into the table), and so on. In other words, the optimizer does not have enough information to generate a query plan. A simple test to find out if something can be used as a bind variable is to ask, "Can I substitute a literal value (a string, an integer—whatever is appropriate) in its place and have SQL*Plus run it legally?" If the answer is yes, then you can use a bind variable there; otherwise, you cannot.

Table 5-2 gives some examples (with explanations) of correct and incorrect uses of ? as a bind variable placeholder.

Table 5-2. *Examples of Valid and Invalid Uses of Bind Variables in Statements*

Statement	Value(s) to Be Bound With	Valid?
`? into t1 values ( ?, ?, ? )`	`insert`	No
Explanation: `insert` is a keyword—you can't use bind variables for keywords. (Try running `'insert' into t1 values (...)` in SQL*Plus.)		
`update emp set job=? where ename = ?`	`CLERK, KING`	Yes
Explanation: You can legally run `update emp set job='CLERK' where ename = 'KING'`.		
`delete emp where ename=?`	`KING`	Yes
Explanation: You can legally run `delete emp where ename='KING'`.		
`select ?, ename from emp where empno=?`	`1, 7629`	Yes
Explanation: You can legally run `select 1, ename from emp where empno=7629`.		
`select ?, ename from emp where empno=?`	`empno, 7629`	No
Explanation: You can bind values but not column names or table names in the `select` clause. If you bind with a constant string value of `empno`, there won't be a runtime exception, though you will get a constant string value of `empno` for all rows returned instead of the value of column `empno`, which is most likely what you intended.		

Update Batching

The update batching feature is relevant for data modification statements such as `insert`, `delete`, and `update`. It allows you to submit multiple data modification statements in one batch, thus saving network round-trips and improving performance significantly in many cases. Oracle supports two models for batch updates:

- The standard model implements the JDBC 2.0 specification and is referred to as *standard update batching*.
- The Oracle-specific model is independent of the JDBC 2.0 specification and is referred to as *Oracle update batching*. Note that to use Oracle update batching, you need to cast the `PreparedStatement` object to the `OraclePreparedStatement` object.

Whether you use standard or Oracle update batching, you must disable autocommit mode. In case an error occurs while you are executing a batch, this gives you the option to commit or roll back the operations that executed successfully prior to the error. This is yet another argument in favor of disabling autocommit.

Types of Statements Supported by Oracle's Batching Implementation

Oracle does not implement true batching for generic `Statement` and `CallableStatement` objects, even though it supports the *use* of standard update batching syntax for these objects. Thus, for Oracle databases using Oracle's JDBC drivers, update batching is relevant only for `PreparedStatement` objects. In other words, only `PreparedStatement` and, by extension, `OraclePreparedStatement` objects can gain performance benefits by using batch updates, regardless of whether we use standard or Oracle update batching. In a `CallableStatement` object, both the connection default batch value and the statement batch value are overridden with a value of 1. In a generic `Statement` object, there is no statement batch value, and the connection default batch value is overridden with a value of 1.

■ **Tip** Whether you use standard update batching or Oracle update batching, you will gain performance benefits only when your code involves `PreparedStatement` objects.

Standard Update Batching

In standard update batching, you manually add operations to the batch and then explicitly choose when to execute the batch. This batching is recommended when JDBC code portability across different databases and JDBC drivers is a higher priority than performance. Instead of using the `executeUpdate()` method to execute an `insert`, `update`, or `delete`, you add an operation to a batch using the `addBatch()` method of the `Statement` interface:

```
public void addBatch( String sql) throws SQLException;
```

At the end, when you want to send the entire batch for execution, you manually execute the batch by invoking the `executeBatch()` method of the `Statement` interface:

```
public int[] executeBatch()  throws SQLException;
```

This method submits the batch of operations to the database for execution, and if they all execute successfully, the database returns an array of update counts. The elements of the returned array are ordered to correspond to the batch commands, which maintain the order in which they were added to the batch. In the Oracle implementation of standard update batching, the values of the array elements are as follows:

- For a prepared statement batch, it is not possible to know the number of rows affected in the database by each individual statement in the batch. Therefore, all array elements have a value of –2 (or the constant `Statement.SUCCESS_NO_INFO`). According to the JDBC 2.0 specification, a value of –2 indicates that the operation was successful but the number of rows affected is unknown.

- For a `Statement` batch or a `CallableStatement` batch, the array contains the actual update counts indicating the number of rows affected by each operation. The actual update counts can be provided because Oracle JDBC cannot use true batching for generic and callable statements in the Oracle implementation of standard update batching, as mentioned earlier.

If one of the batched operations fails during an `executeBatch()` call, then execution stops and a `java.sql.BatchUpdateException` (a subclass of `java.sql.SQLException`) is thrown. After a batch exception, the update counts array can be retrieved using the `getUpdateCounts()` method of the `BatchUpdateException` object. This returns an `int` array of update counts, just as the `executeBatch()` method does, the contents of which are as follows:

- For a prepared statement batch, each element has a value of –3 (or `Statement.EXECUTE_FAILED`), indicating that an operation did not complete successfully. In this case, presumably just one operation actually failed, but because the JDBC driver does not know which operation that was, it labels *all* the batched operations as failures.
- For a generic statement batch or callable statement batch, the update counts array is only a partial array that contains the actual update counts up to the point of the error. The actual update counts can be provided because Oracle JDBC *cannot* use true batching for generic and callable statements in the Oracle implementation of standard update batching.

If you want to clear the current batch of operations instead of executing it, simply use the `clearBatch()` method of the `Statement` interface. A `clearBatch()` essentially resets the batch contents to empty.

```
public void clearBatch();
```

Standard Update Batching Example

First, we create a simple table, `t1`, with one column, `x`, which is also the primary key:

```
scott@ORA10G> create table t1
  2  (
  3    x number primary key
  4  );
Table created.
```

The class `DemoStandardBatching` illustrates standard update batching:

```
/* This program illustrates the use of standard update batching.
* COMPATIBLITY NOTE: runs successfully against 10.1.0.2.0. and  9.2.0.1.0.
*/
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.BatchUpdateException;
import java.sql.PreparedStatement;
import java.sql.Statement; // for accessing constants only
import book.util.JDBCUtil;
import book.util.Util;
class DemoStandardUpdateBatching
{
  public static void main(String args[])
  {
```

Inside `main()`, we first validate program arguments, declare variables, and obtain a connection in the `try catch` block. Recall that `autocommit` is set to `false` in the `JDBCUtil.getConnection()` method:

```
Util.checkProgramUsage( args );
Connection conn = null;
PreparedStatement pstmt = null;
int[] updateCounts = null;
try
{
  // get connection, set autocommit to false in JDBCUtil method
  // Note: setting autocommit to false is required,
  // especially when you are using update batching.
  // of course, you should do this anyway for
  // transaction integrity and performance, especially
  // when developing applications on Oracle.
  conn = JDBCUtil.getConnection("benchmark", "benchmark", args[0]);
```

We prepare an `insert` statement next:

```
  // prepare a statement to insert data
  pstmt = conn.prepareStatement( "insert into t1( x ) values ( ? )");
```

The batching begins now. Instead of executing the statement after binding it with different values, we add it to a batch. We add three inserts to the batch, each with different values for column `x`:

```
  // first insert
  pstmt.setInt(1, 1 );
  pstmt.addBatch();
  // second insert
  pstmt.setInt(1, 2 );
  pstmt.addBatch();
  // third insert
  pstmt.setInt(1, 3 );
  pstmt.addBatch();
```

We then send the batch of three `insert` statements to be executed in one shot by using the `sendBatch()` method. The method returns an array of update counts. In the case of success, this count gives us the total number of successful insert operations. We conclude the transaction by committing it:

```
  // Manually execute the batch
  updateCounts = pstmt.executeBatch();
  System.out.println( "Inserted " + updateCounts.length + " rows successfully");
  conn.commit();
}
```

In case one of the `insert` operations fails, a `BatchUpdateException` is thrown. We handle this exception by obtaining the update count array from the exception object and printing out

the values. In Oracle, there is not much value in the logic of this loop since it does not tell us which operation failed; it tells us only that *one* of the operations failed. After the loop, we print the exception and roll back:

```
    catch (BatchUpdateException e)
    {
      // Check if each of the statements in batch was
      // successful - if not, throw Exception
      updateCounts = e.getUpdateCounts();
      for( int k=0; k < updateCounts.length; k++ )
      {
        /*
           For a standard prepared statement batch, it is impossible
           to know the number of rows affected in the database by
           each individual statement in the batch.
           According to the JDBC 2.0 specification, a value of
           Statement.SUCCESS_NO_INFO indicates that the operation
           was successful but the number of rows affected is unknown.
         */
        if( updateCounts[k] != Statement.SUCCESS_NO_INFO )
        {
          String message = "Error in standard batch update - Found a value" +
            " of " + updateCounts[k] + " in the update count "+
            "array for statement number " + k;
          System.out.println( message );
        }
      }
      // print the exception error message and roll back
      JDBCUtil.printExceptionAndRollback( conn, e );
    }
```

At the end of the class, we have the standard handling of the generic exception and the `finally` clause to release JDBC resources:

```
    catch (Exception e)
    {
      // handle the generic exception; print error message and roll back
      JDBCUtil.printExceptionAndRollback( conn, e );
    }
    finally
    {
      // release JDBC resource in the finally clause.
      JDBCUtil.close( pstmt );
      JDBCUtil.close( conn );
    }
  } // end of main
} // end of program
```

When we execute the preceding program with table t1 empty, we get the following output:

```
B:\>java DemoStandardUpdateBatching ora10g
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(PORT=1521)(HOST=rmeno
n-lap))(CONNECT_DATA=(SID=ora10g)))
Inserted 3 rows successfully
```

If we execute it again, we will get an error, because x is a primary key:

```
B:\>java DemoStandardUpdateBatching ora10g
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(PORT=1521)(HOST=rmeno
n-lap))(CONNECT_DATA=(SID=ora10g)))
Error in standard batch update - Found a value of -3 in the update count array f
or statement number 0
Error in standard batch update - Found a value of -3 in the update count array f
or statement number 1
Error in standard batch update - Found a value of -3 in the update count array f
or statement number 2
Exception caught! Exiting ..
error message: ORA-00001: unique constraint (BENCHMARK.SYS_C005873) violated

java.sql.BatchUpdateException: ORA-00001: unique constraint (BENCHMARK.SYS_C0058
73) violated…
```

Regardless of how many rows result in an error, the Oracle JDBC driver puts a value of –3 (indicating failure) in the update count for all of the rows.

Let's now turn our attention to Oracle update batching.

Oracle Update Batching

With Oracle update batching, the first step is to define a *batch value*, which is the number of operations you want to process per round-trip. You can set this batch value in two ways:

- By invoking the `setDefaultExecuteBatch()` method on the `OracleConnection` object:

  ```
  public void setDefaultExecuteBatch(int);
  ```

  This sets the batch size on all the statements associated with the connection to the specified value. As you may have guessed, there is a corresponding `getDefault➥ExecuteBatch()` method available in the `OracleConnection` interface as well.

- By invoking the `setExecuteBatch()` method on the `OraclePreparedStatement` object:

  ```
  public void setExecuteBatch(int);
  ```

  This sets the batch size on a particular statement and is usually the way applications use the batching feature. A corresponding `getExecuteBatch()` method is available in the `OraclePreparedStatement` interface as well. Remember that the statement-level batch overrides the one set at the connection level.

If you want to explicitly execute accumulated operations before the batch value in effect is reached, then you can use the `sendBatch()` method of the `OraclePreparedStatement` interface, which returns the number of operations successfully completed:

```
public int sendBatch(int);
```

Just as in the case of standard update batching, you can clear the current batch of operations instead of executing it by using the `clearBatch()` method of the `Statement` interface. A `clearBatch()` essentially resets the batch contents to empty.

```
public void clearBatch();
```

Oracle Update Batching Example

In this section, we'll look at an example that illustrates Oracle update batching. The `DemoOracleUpdateBatching` class begins with the import statements and the `main()` method declaration. Within the `main()` method, we obtain the connection as usual:

```
/* This program illustrates use of Oracle update batching.
* COMPATIBLITY NOTE: runs successfully against 10.1.0.2.0. and 9.2.0.1.0.
*/
import java.sql.SQLException;
import java.sql.Statement; // for accessing constants only
import oracle.jdbc.OraclePreparedStatement;
import oracle.jdbc.OracleConnection;
import book.util.JDBCUtil;
import book.util.Util;
class DemoOracleUpdateBatching
{
  public static void main(String args[])
  {
    Util.checkProgramUsage( args );
    OracleConnection oconn = null;
    OraclePreparedStatement opstmt = null;
    try
    {
      // get connection, set it to autocommit within JDBCUtil.getConnection()
      oconn = (OracleConnection)JDBCUtil.getConnection(
        "benchmark", "benchmark", args[0]);
```

We prepare an `insert` statement, casting the returned object to the `OraclePreparedStatement` interface:

```
      // prepare a statement to insert data
      opstmt = (OraclePreparedStatement) oconn.prepareStatement(
        "insert into t1( x ) values ( ? )");
```

We set the batch size to `3` at the statement level:

```
      opstmt.setExecuteBatch( 3 );
```

We then insert three rows, printing out the number of rows returned each time. Since the batch size is 3, Oracle queues up the batches and executes them all together with the third insert.

```
  // first insert
  opstmt.setInt(1, 1 );
  // following insert is queued for execution by JDBC
  int numOfRowsInserted = opstmt.executeUpdate();
  System.out.println("num of rows inserted: " + numOfRowsInserted );
  // second insert
  opstmt.setInt(1, 2 );
  // following insert is queued for execution by JDBC
  numOfRowsInserted = opstmt.executeUpdate();
  System.out.println("num of rows inserted: " + numOfRowsInserted );
  // third insert
  opstmt.setInt(1, 3 );
  // since batch size is 3, the following insert will result
  // in JDBC sending all three inserts queued so far (including
  // the one below) for execution
  numOfRowsInserted = opstmt.executeUpdate();
  System.out.println("num of rows inserted: " + numOfRowsInserted );
```

We next insert another row. This insert will get queued again in a fresh batch.

```
  // fourth insert
  opstmt.setInt(1, 4 );
  // following insert is queued for execution by JDBC
  numOfRowsInserted = opstmt.executeUpdate();
  System.out.println("num of rows inserted: " + numOfRowsInserted );
```

We send this batch explicitly using the `sendBatch()` method:

```
  // now if you want to explicitly send the batch, you can
  // use the sendBatch() method as shown below.
  numOfRowsInserted = opstmt.sendBatch();
  System.out.println("num of rows sent for batch: " + numOfRowsInserted );
```

Finally, we commit our transaction and end the program:

```
  oconn.commit();
}
catch (Exception e)
{
  // handle the exception properly - in this case, we just
  // print a message and roll back
  JDBCUtil.printExceptionAndRollback( oconn, e );
}
finally
{
  // close the result set, statement, and connection.
```

```
      // ignore any exceptions since we are in the
      // finally clause.
      JDBCUtil.close( opstmt );
      JDBCUtil.close( oconn );
    }
  }
}
```

When we execute the preceding program (after deleting any pre-existing rows from `t1`), we get the following output:

```
B:\>java DemoOracleUpdateBatching ora10g
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(PORT=1521)(HOST=rmeno
n-lap))(CONNECT_DATA=(SID=ora10g)))
num of rows inserted: 0
num of rows inserted: 0
num of rows inserted: 3
num of rows inserted: 0
num of rows sent for batch: 1
```

As expected, the first two inserts were actually queued up and sent along with the third insert, as is evident from the number of rows inserted. The next insert is, however, executed explicitly when we use the `sendBatch()` method.

Now that you're familiar with both types of batching, next we'll cover a caveat regarding mixing interdependent statements in a batch.

Mixing Interdependent Statements in a Batch

Both update batching implementations generally work as expected. There are cases where the results may be surprising to you (especially when you use Oracle update batching). Intuitively, you would expect that changing the batch size should impact only the performance of the application, not the actual data inserted, deleted, and so on. However, this is not the case if you have multiple statements using batching in a loop, and some of these statements can have an impact on the rows manipulated by other statements in the loop.

Consider the following class (based on a test case supplied by Tom Kyte), which uses Oracle update batching. It takes a batch size as a command-line parameter and uses Oracle update batching to set the execution batch size on an insert and a delete from the same table `t1` that we created in the earlier examples:

```
/* This program illustrates a special case of Oracle update batching
   where the results are nonintuitive although correct as per
   the JDBC specification.
* COMPATIBLITY NOTE: runs successfully against 10.1.0.2.0. and  9.2.0.1.0.
*/
import java.sql.Statement;
import java.sql.Connection;
import java.sql.PreparedStatement;
import oracle.jdbc.OraclePreparedStatement;
import book.util.JDBCUtil;
```

```
class TestUpdateBatching
{
  public static void main(String args[])throws Exception
  {
    if( args.length != 1 )
    {
      System.out.println("Usage: java TestUpdateBatching <batch_size>" );
    }
    int batchSize = Integer.parseInt( args[0] );
    Connection conn = null;
    Statement stmt = null;
    OraclePreparedStatement ipstmt = null;
    OraclePreparedStatement dpstmt = null;
    try
    {
      conn = JDBCUtil.getConnection("benchmark", "benchmark", "ora10g");
      stmt = conn.createStatement ();
      ipstmt = (OraclePreparedStatement) conn.prepareStatement(
         "insert into t1( x ) values ( ? )");
      ipstmt.setExecuteBatch( batchSize );
      dpstmt = (OraclePreparedStatement) conn.prepareStatement(
        "delete from t1 where x = ?" );
      dpstmt.setExecuteBatch( batchSize );
```

After creating the `insert` and `delete` statements and setting their batch size, we go in a loop where the `insert` statement inserts the loop index `i`, and the `delete` statement deletes the values matching 1 added to the loop index (i.e., `i + 1`).

```
      for( int i = 0; i < 2; i++ )
      {
        ipstmt.setInt(1, i );
        int numOfRowsInserted = ipstmt.executeUpdate();
        System.out.println("num of rows inserted: " + numOfRowsInserted );
        dpstmt.setInt(1, i+1 );
        int numOfRowsDeleted = dpstmt.executeUpdate();
        System.out.println("num of rows Deleted: " + numOfRowsDeleted );
      }
```

We send the batches for any remaining rows outside and commit the transaction at the end of the program:

```
      ipstmt.sendBatch();
      dpstmt.sendBatch();
      conn.commit();
    }
    catch (Exception e)
    {
      // handle the exception properly - in this case, we just
      // print a message and roll back
      JDBCUtil.printExceptionAndRollback( conn, e );
```

```
    }
    finally
    {
      // close the result set, statement, and connection.
      // ignore any exceptions since we are in the
      // finally clause.
      JDBCUtil.close( ipstmt );
      JDBCUtil.close( dpstmt );
      JDBCUtil.close( conn );
    }
  }
}
```

When we run the program with a batch size of 1, we get the following output:

```
B:\>java TestUpdateBatching 1
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(PORT=1521)(HOST=rmeno
n-lap))(CONNECT_DATA=(SID=ora10g)))
num of rows inserted: 1
num of rows Deleted: 0
num of rows inserted: 1
num of rows Deleted: 0
```

A select from table `t1` gives

```
benchmark@ORA10G> select * from t1;
         0
         1
```

After deleting all rows from table `t1`, let's run the program with a batch size of 2:

```
B:\>java TestUpdateBatching 2
URL:jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(PORT=1521)
(HOST=rmenon-lap))(CONNECT_DATA=(SID=ora10g)))
num of rows inserted: 0
num of rows Deleted: 0
num of rows inserted: 2
num of rows Deleted: 1
```

When we do a select this time, we get a different result from the one we got when the batch size was 1:

```
benchmark@ORA10G> select * from t1;
         0
```

This is a case in which changing the batch size apparently changed the program's outcome! Although this looks like a bug, it turns out that this behavior is correct. Recall that in Oracle update batching, a batch is automatically sent to the database once the batch size is reached. When the batch size is 1, the `delete` statement does not affect any rows, since it attempts to delete values that do not exist in the database. So, we get two rows as expected. Table 5-3 lists the steps the JDBC driver goes through when the batch size is 2.

Table 5-3. *Steps in the* `for` *Loop When the Batch Size is 2 While Executing the* `TestUpdateBatching` *Class*

Loop Index Value (Value of `i`)	Statement	What Happens
0	`insert`	The batch size is 2. The statement gets queued up.
	`delete`	Since this is a new statement with a batch size of 2, it also gets queued up.
2	`insert`	This is the second `insert`, which implies we have reached the batch size limit. The JDBC driver sends both inserts inserting two rows, with column values of 0 and 1 for column `x` in table `t1`.
	`delete`	This is the second delete, which implies we have reached the batch size limit. The JDBC driver sends and executes both deletes. The first delete deletes the value of a row with a column value of 1. The second delete does not delete any rows since no rows match the criteria. Hence, we are left with just one row with a value of 0 for column `x`.

The key thing to note is that the `delete` statements in the loop directly affect the values inserted by the `insert` statement. In the case of a batch size of 1, the deletes worked on the data available after the inserts had been applied to the database. In the case of a batch size of 2, the state of the database on which deletes worked was different (since all deletes were sent after the two inserts were executed, not just the preceding ones). Since in the case of Oracle update batching this happens implicitly, it looks more confusing.

From this discussion, we can conclude that the batch size can impact the results when we mix different statements where the following statements affect the results of the preceding ones. In such cases, we should either ensure that our logic does not get impacted by the batch size or avoid using batches altogether. For example, we can change the loop structure of the program `TestUpdateBatching` to the following to get consistent results (assuming we want all inserts applied before all deletes):

```
for( int i = 0; i < 2; i++ )
{
  ipstmt.setInt(1, i );
  int numOfRowsInserted = ipstmt.executeUpdate();
  System.out.println("num of rows inserted: " + numOfRowsInserted );
}
ipstmt.sendBatch();
for( int i = 0; i < 2; i++ )
{
  dpstmt.setInt(1, i+1 );
  int numOfRowsDeleted = dpstmt.executeUpdate();
  System.out.println("num of rows Deleted: " + numOfRowsDeleted );
}
dpstmt.sendBatch();
conn.commit();
```

Oracle Update Batching vs. Standard Update Batching

In Oracle update batching, as soon as the number of statements added to the batch reaches the batch value, the batch is executed. Recall that

- You can set a default batch at the `Connection` object level, which applies to any prepared statement executed using that connection.
- You can set a statement batch value for any individual prepared statement. This value overrides any batch value set at the `Connection` object level.
- You can explicitly execute a batch at any time, overriding both the connection batch value and the statement batch value.

In contrast to Oracle update batching, standard update batching involves an explicit manual execution of the batch, as there is no batch value. You should choose standard update batching if you are concerned about the portability of your Java code across databases. Otherwise, choose Oracle update batching because you get better performance out of it (as you'll learn in the next section).

■ **Caution** You can't mix and match the standard update batching syntax and Oracle update batching syntax. If you do so, you will get a `SQLException`.

Batching Performance Analysis

The following `StandardVsOracleBatching` class compares the elapsed time and the latches consumed for inserting 10,000 records for the case of standard update batching and Oracle update batching. The `StandardVsOracleBatching` class uses the utility class (discussed in the section "JDBC Wrapper for RUNSTATS" of Chapter 1) to compare the latches consumed and the time taken for the preceding three cases.

After the imports, the class begins by declaring some private variables:

```
/* This program compares standard update batching with Oracle update
    batching for elapsed times and latches consumed using the
    JRunstats utility.
* COMPATIBLITY NOTE: runs successfully against 10.1.0.2.0. and  9.2.0.1.0.
*/
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import oracle.jdbc.OraclePreparedStatement;
import book.util.JDBCUtil;
import book.util.JRunstats;
class StandardVsOracleBatching
{
  private static int s_numberOfRecords = 0;
```

```
private static int s_batchSize = 1;
private static long s_start = 0;
private static long s_middle = 0;
private static long s_end = 0;
private static int[] s_batchSizeArr =
  { 1, 5, 10, 50, 75, 100, 150, 200, 300, 400, 500,
    750, 1000, 2000, 3000, 5000, 10000 };
```

In particular, the static variable `s_batchSizeArr` declares an array of batch sizes that we will run our three cases with. The following `_checkUsage()` method simply checks the program usage. The program takes the total number of records that we want to insert as a command-line parameter:

```
private static void _checkUsage (String[] args)
{
  int argc = args.length;
  if( argc != 1)
  {
    System.err.println(
      "Usage: java StandardVsOracleBatching <number of records>" );
    Runtime.getRuntime().exit(1);
  }
  s_numberOfRecords = Integer.parseInt( args[0] );
}
```

In the `main()` method, we invoke `_checkUsage()` and get the connection in the `try catch` block after declaring some variables:

```
public static void main(String args[])
  public static void main(String args[])
{
  _checkUsage( args );
  Connection conn = null;
  PreparedStatement pstmt = null;
  OraclePreparedStatement opstmt = null;
  String insertStmtStr = "insert into t1( x, y ) values ( ?, ?)";
  try
  {
    // get connection; set autocommit to false within JDBCUtil.
    conn = JDBCUtil.getConnection("benchmark", "benchmark", "ora10g");
```

We prepare the statements to be used for standard and Oracle update batching, respectively:

```
    pstmt = conn.prepareStatement( insertStmtStr );
    opstmt = (OraclePreparedStatement) conn.prepareStatement( insertStmtStr );
```

For each batch size, we execute the two cases, beginning with the case of standard update batching:

```
for(int x=0; x < s_batchSizeArr.length; x++ )
{
```

In the loop, we first prepare the benchmark statements in `JRunstats`:

```
JRunstats.prepareBenchmarkStatements( conn );
```

We set the current batch size in a variable:

```
s_batchSize = s_batchSizeArr[x];
```

Then we mark the beginning of the execution of the inserts based on standard update batching. We also mark our start time.

```
// mark beginning of execute with standard update batching
JRunstats.markStart( conn );
s_start = System.currentTimeMillis();
```

The following `for` loop inserts 10,000 records using the current batch size:

```
// execute with standard update batching
for( int i=0; i < s_numberOfRecords; i++)
{
  // batch s_batchSize number of statements
  // before sending them as one round-trip.
  int j = 0;
  for( j=0; j < s_batchSize; j++)
  {
    pstmt.setInt(1, i );
    pstmt.setString(2, "data" + i  );
    pstmt.addBatch();
   // System.out.println( "Inserted " + numOfRowsInserted + " row(s)" );
  }
  i += (j-1);
  int[] updateCounts = pstmt.executeBatch();
  //System.out.println( "i = " + i );
}
```

Next, we insert the same number of records using Oracle update batching after marking the middle of our benchmark:

```
// mark beginning of execute with Oracle update batching
JRunstats.markMiddle( conn );
s_middle = System.currentTimeMillis();
// set the execute batch size
opstmt.setExecuteBatch( s_batchSize );
// bind the values
for( int i=0; i < s_numberOfRecords; i++)
{
```

```
          // bind the values
          opstmt.setInt(1, i );
          opstmt.setString(2, "data"+i  );
          int numOfRowsInserted  = opstmt.executeUpdate();
        }
```

We mark the end of the benchmark run, and then we print out the results followed by various `close()` statements being invoked to release JDBC resources in the `finally` clause:

```
        s_end = System.currentTimeMillis();
        JRunstats.markEnd( conn, 10000 );
        System.out.println( "Standard Update batching (recs="+
          s_numberOfRecords+ ", batch=" + s_batchSize + ") = "
          + (s_middle - s_start ) + " ms" );
        System.out.println( "Oracle Update batching (recs="+
          s_numberOfRecords+ ", batch=" + s_batchSize + ") = " +
          (s_end - s_middle ) + " ms");
        conn.commit();
        JRunstats.closeBenchmarkStatements( conn );
      }
    }
    catch (Exception e)
    {
      // handle the exception properly - in this case, we just
      // print a message and roll back
      JDBCUtil.printExceptionAndRollback( conn, e );
    }
    finally
    {
      // release JDBC resources in the finally clause.
      JDBCUtil.close( pstmt );
      JDBCUtil.close( opstmt );
      JDBCUtil.close( conn );
    }
  }
}
```

I ran the `StandardVsOracleBatching` program with multiple values of batch sizes for inserting 10,000 records into table `t1` that I created as follows:

```
benchmark@ORA10G> create table t1 ( x number,  y varchar2(20));
Table created.
```

Table 5-4 and Figure 5-3 show the results of comparing elapsed times taken when I inserted 10,000 records for the cases of standard update batching and Oracle update batching. For no batching, it took an average of 13,201 milliseconds to insert 10,000 records.

Table 5-4. *Comparing Standard Update Batching with Oracle Update Batching for Inserting 10,000 Records*

Batch Size	Standard Update Batching (Milliseconds)	Oracle Update Batching (Milliseconds)
1	13,201	13,201
5	2,861	2,629
10	1,554	1,406
30	702	613
50	532	440
75	456	359
100	402	322
150	370	304
200	278	274
300	349	280
400	345	269
500	320	263
750	345	282
1,000	327	232
2,000	328	236
3,000	406	207
5,000	372	375
10,000	490	461

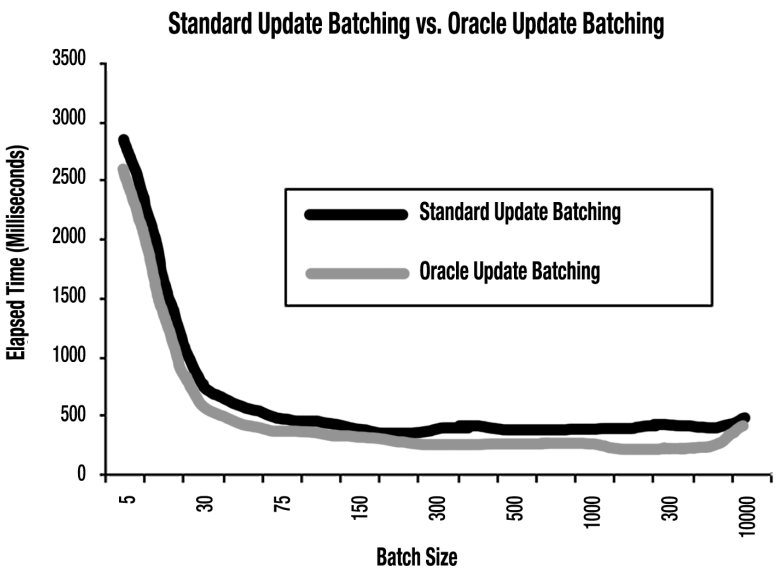

Figure 5-3. *Comparing standard update batching with Oracle update batching for inserting 10,000 records*

Figure 5-4 shows a comparison of latches consumed for standard versus Oracle update batching. The case of no batching is shown as batch size 1.

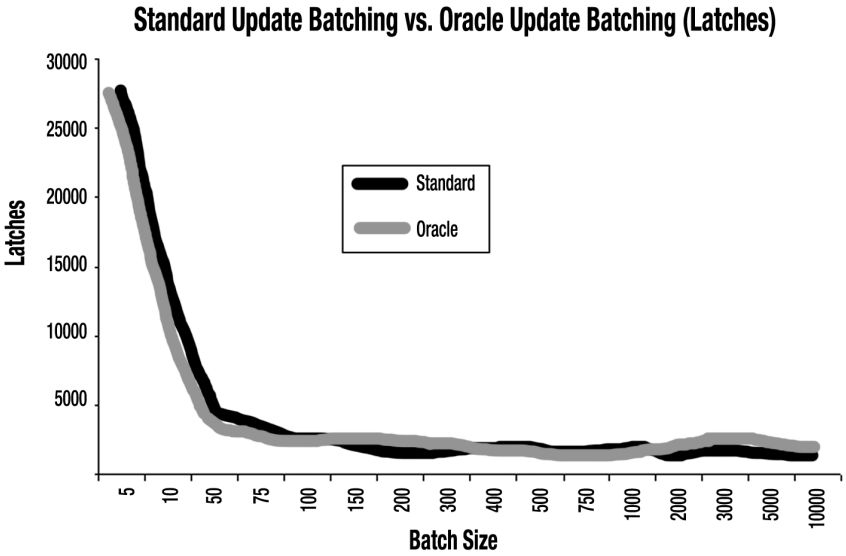

Figure 5-4. *Comparing standard update batching with Oracle update batching in terms of latches consumed*

We can make the following observations from the charts in Figure 5-3 and Figure 5-4:

- Batching makes a *huge* difference, both in elapsed time and in latches consumed. Without batching enabled (or equivalently with a batch size of 1), the average time taken is 13,201 milliseconds compared to the worst case of 2,861 milliseconds and the best case of 207 milliseconds when batching is used. Similarly, the latches consumed go down dramatically once batching is used, as shown in Figure 5-4.
- In general, in terms of elapsed time, Oracle update batching performs better than standard update batching. In the preceding experiment, standard update batching took between 1% to 30% more time as compared to Oracle update batching, depending on the batch size.
- In terms of latch consumption, the difference between Oracle update batching and standard update batching seems to be negligible.
- Although Oracle documentation recommends a batch size of 5 to 30 as optimal, the preceding case shows that a batch size of around 200 was best for standard update batching, and a batch size of around 3,000 was best for Oracle update batching in terms of elapsed times. In terms of latch consumption, a batch size of 2,000 was best for standard update batching, whereas a batch size of 750 resulted in the lowest number of latches consumed for Oracle update batching. Of course, this does not prove that these batch sizes are always optimal—it only demonstrates that you should benchmark critical portions of code using an experiment such as that just presented to find out what the optimal batch size is in your case.

- Notice that as the batch size increases, you gain in terms of elapsed time initially, and then you start witnessing a negative effect. For example, in the case of Oracle update batching, after a batch size of 3,000 you start seeing decreasing performance. This indicates that you cannot blindly set the batch size to the maximum number of records you modify; rather, you have to benchmark your particular scenario to arrive at an optimal batch size.
- It is a good idea to parameterize the batch size for an important set of operations so that you can easily change it.

Summary

In this chapter, you learned how to query and modify data using the JDBC classes `Statement`, `PreparedStatement`, and `OraclePreparedStatement`. You also learned the reasons you should not use the `Statement` class in production code, as it does not support using bind variables. By using bind variables in your program, you not only make your application more performant and scaleable, but you also make it more secure by preventing SQL injection attacks.

You discovered how to boost application performance tremendously by using update batching, which in Oracle is available only when you use prepared statements. You saw a comparison of standard and Oracle update batching in terms of elapsed times and latch consumption, and you observed that Oracle update batching outperforms standard update batching in terms of elapsed time (in terms of latches, the difference between the two is negligible). You also saw how batching can dramatically reduce the latch consumption, thus improving scalability. You looked at how mixing interdependent batch statements in your application can sometimes lead to unexpected results.

A major take-away message from this chapter is that, if you are embedding SQL statements in your JDBC code, you should use a `PreparedStatement` object and use bind variables wherever appropriate. However, a strong case can also be made for wrapping your DML statements in PL/SQL packaged procedures and invoking them from JDBC using `CallableStatement` objects. This is what we will examine in the next chapter.