Accessing Databases with JDBC

Topics in This Chapter

- Connecting to databases: the seven basic steps
- Simplifying JDBC usage: some utilities
- Using precompiled (parameterized) queries
- Creating and executing stored procedures
- Updating data through transactions
- Using JDO and other object-to-relational mappings

Chapter 17

JDBC provides a standard library for accessing relational databases. By using the JDBC API, you can access a wide variety of SQL databases with exactly the same Java syntax. It is important to note that although the JDBC API standardizes the approach for connecting to databases, the syntax for sending queries and committing transactions, and the data structure representing the result, JDBC does *not* attempt to standardize the SQL syntax. So, you can use any SQL extensions your database vendor supports. However, since most queries follow standard SQL syntax, using JDBC lets you change database hosts, ports, and even database vendors with minimal changes to your code.

DILBERT reprinted by permission of United Feature Syndicate, Inc.

Officially, JDBC is not an acronym and thus does not stand for anything. Unofficially, "Java DataBase Connectivity" is commonly used as the long form of the name.

Although a complete tutorial on database programming is beyond the scope of this chapter, we cover the basics of using JDBC in Section 17.1 (Using JDBC in General), presuming you are already familiar with SQL.

After covering JDBC basics, in Section 17.2 (Basic JDBC Examples) we present some JDBC examples that access a Microsoft Access database.

To simplify the JDBC code throughout the rest of the chapter, we provide some utilities for creating connections to databases in Section 17.3 (Simplifying Database Access with JDBC Utilities).

In Section 17.4 (Using Prepared Statements), we discuss prepared statements, which let you execute similar SQL statements multiple times; this can be more efficient than executing a raw query each time.

In Section 17.5 (Creating Callable Statements), we examine callable statements. Callable statements let you execute database stored procedures or functions.

In Section 17.6 (Using Database Transactions), we cover transaction management for maintaining database integrity. By executing changes to the database within a transaction, you can ensure that the database values are returned to their original state if a problem occurs.

In Section 17.7, we briefly examine object-to-relational mapping (ORM). ORM frameworks provide a complete object-oriented approach to manage information in a database. With ORM, you simply call methods on objects instead of directly using JDBC and SQL.

For advanced JDBC topics including accessing databases with custom JSP tags, using data sources with JNDI, and increasing performance by pooling database connections, see Volume 2 of this book. For more details on JDBC, see http://java.sun.com/products/jdbc/, the online API for `java.sql`, or the JDBC tutorial at http://java.sun.com/docs/books/tutorial/jdbc/.

17.1 Using JDBC in General

In this section we present the seven standard steps for querying databases. In Section 17.2 we give two simple examples (a command-line program and a servlet) illustrating these steps to query a Microsoft Access database.

Following is a summary; details are given in the rest of the section.

1. **Load the JDBC driver.** To load a driver, you specify the classname of the database driver in the `Class.forName` method. By doing so, you automatically create a driver instance and register it with the JDBC driver manager.

2. **Define the connection URL.** In JDBC, a connection URL specifies the server host, port, and database name with which to establish a connection.
3. **Establish the connection.** With the connection URL, username, and password, a network connection to the database can be established. Once the connection is established, database queries can be performed until the connection is closed.
4. **Create a `Statement` object.** Creating a `Statement` object enables you to send queries and commands to the database.
5. **Execute a query or update.** Given a `Statement` object, you can send SQL statements to the database by using the `execute`, `executeQuery`, `executeUpdate`, or `executeBatch` methods.
6. **Process the results.** When a database query is executed, a `ResultSet` is returned. The `ResultSet` represents a set of rows and columns that you can process by calls to `next` and various `getXxx` methods.
7. **Close the connection.** When you are finished performing queries and processing results, you should close the connection, releasing resources to the database.

Load the JDBC Driver

The driver is the piece of software that knows how to talk to the actual database server. To load the driver, you just load the appropriate class; a `static` block in the driver class itself automatically makes a driver instance and registers it with the JDBC driver manager. To make your code as flexible as possible, avoid hard-coding the reference to the classname. In Section 17.3 (Simplifying Database Access with JDBC Utilities) we present a utility class to load drivers from a `Properties` file so that the classname is not hard-coded in the program.

These requirements bring up two interesting questions. First, how do you load a class without making an instance of it? Second, how can you refer to a class whose name isn't known when the code is compiled? The answer to both questions is to use `Class.forName`. This method takes a string representing a fully qualified classname (i.e., one that includes package names) and loads the corresponding class. This call could throw a `ClassNotFoundException`, so it should be inside a `try/catch` block as shown below.

```
try {
  Class.forName("connect.microsoft.MicrosoftDriver");
  Class.forName("oracle.jdbc.driver.OracleDriver");
  Class.forName("com.sybase.jdbc.SybDriver");
} catch(ClassNotFoundException cnfe) {
  System.err.println("Error loading driver: " + cnfe);
}
```

One of the beauties of the JDBC approach is that the database server requires no changes whatsoever. Instead, the JDBC driver (which is on the client) translates calls written in the Java programming language into the native format required by the server. This approach means that you have to obtain a JDBC driver specific to the database you are using and that you will need to check the vendor's documentation for the fully qualified class name to use.

In principle, you can use `Class.forName` for any class in your `CLASSPATH`. In practice, however, most JDBC driver vendors distribute their drivers inside JAR files. So, during development be sure to include the path to the driver JAR file in your `CLASSPATH` setting. For deployment on a Web server, put the JAR file in the WEB-INF/lib directory of your Web application (see Chapter 2, "Server Setup and Configuration"). Check with your Web server administrator, though. Often, if multiple Web applications are using the same database drivers, the administrator will place the JAR file in a common directory used by the server. For example, in Apache Tomcat, JAR files common to multiple applications can be placed in *install_dir*/common/lib.

Core Note

You can place your JDBC driver file (JAR file) in the WEB-INF/lib directory for deployment of your application. However, the administrator may choose to move the JAR file to a common library directory on the server.

Figure 17–1 illustrates two common JDBC driver implementations. The first approach is a JDBC-ODBC bridge, and the second approach is a pure Java implementation. A driver that uses the JDBC-ODBC bridge approach is known as a Type I driver. Since many databases support Open DataBase Connectivity (ODBC) access, the JDK includes a JDBC-ODBC bridge to connect to databases. However, you should use the vendor's pure Java driver, if available, because the JDBC-ODBC driver implementation is slower than a pure Java implementation. Pure Java drivers are known as Type IV. The JDBC specification defines two other driver types, Type II and Type III; however, they are less common. For additional details on driver types, see http://java.sun.com/products/jdbc/driverdesc.html.

In the initial examples in this chapter, we use the JDBC-ODBC bridge, included with JDK 1.4, to connect to a Microsoft Access database. In later examples we use pure Java drivers to connect to MySQL and Oracle9i databases.

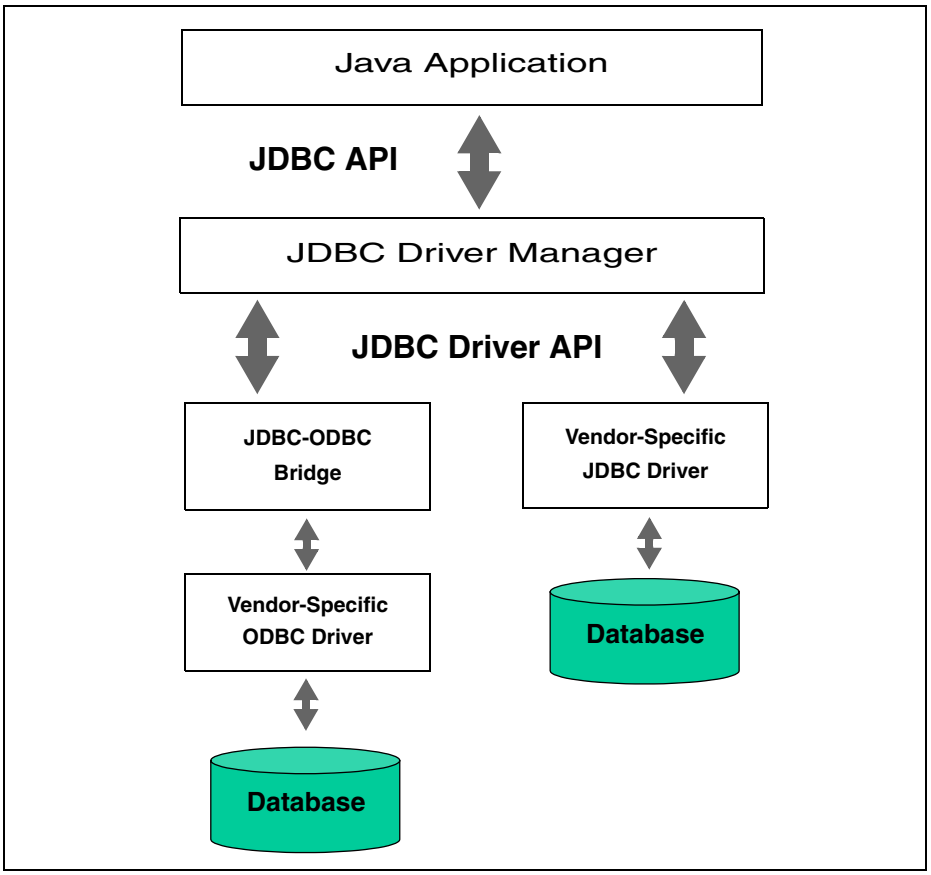

Figure 17–1 Two common JDBC driver implementations. JDK 1.4 includes a JDBC-ODBC bridge; however, a pure JDBC driver (provided by the vendor) yields better performance.

In Section 18.1 (Configuring Microsoft Access for Use with JDBC), we provide driver information for Microsoft Access. Driver information for MySQL is provided in Section 18.2 (Installing and Configuring MySQL), and driver information for Oracle is provided in Section 18.3 (Installing and Configuring Oracle9i Database). Most other database vendors supply free JDBC drivers for their databases. For an up-to-date list of these and third-party drivers, see http://industry.java.sun.com/products/jdbc/drivers/.

Define the Connection URL

Once you have loaded the JDBC driver, you must specify the location of the database server. URLs referring to databases use the `jdbc:` protocol and embed the server host, port, and database name (or reference) within the URL. The exact format is defined in the documentation that comes with the particular driver, but here are a few representative examples.

```
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host +
                   ":" + port + ":" + dbName;
String sybaseURL = "jdbc:sybase:Tds:" + host +
                   ":" + port + ":" + "?SERVICENAME=" + dbName;
String msAccessURL = "jdbc:odbc:" + dbName;
```

Establish the Connection

To make the actual network connection, pass the URL, database username, and database password to the `getConnection` method of the `DriverManager` class, as illustrated in the following example. Note that `getConnection` throws an `SQLException`, so you need to use a `try/catch` block. We're omitting this block from the following example since the methods in the following steps throw the same exception, and thus you typically use a single `try/catch` block for all of them.

```
String username = "jay_debesee";
String password = "secret";
Connection connection =
  DriverManager.getConnection(oracleURL, username, password);
```

The `Connection` class includes other useful methods, which we briefly describe below. The first three methods are covered in detail in Sections 17.4–17.6.

- **`prepareStatement`.** Creates precompiled queries for submission to the database. See Section 17.4 (Using Prepared Statements) for details.
- **`prepareCall`.** Accesses stored procedures in the database. For details, see Section 17.5 (Creating Callable Statements).
- **`rollback/commit`.** Controls transaction management. See Section 17.6 (Using Database Transactions) for details.
- **`close`.** Terminates the open connection.
- **`isClosed`.** Determines whether the connection timed out or was explicitly closed.

An optional part of establishing the connection is to look up information about the database with the `getMetaData` method. This method returns a `DatabaseMetaData` object that has methods with which you can discover the name and version of the database itself (`getDatabaseProductName`, `getDatabaseProductVersion`) or of the JDBC driver (`getDriverName`, `getDriverVersion`). Here is an example.

```
DatabaseMetaData dbMetaData = connection.getMetaData();
String productName =
  dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);
String productVersion =
  dbMetaData.getDatabaseProductVersion();
System.out.println("Version: " + productVersion);
```

Create a Statement Object

A `Statement` object is used to send queries and commands to the database. It is created from the `Connection` using `createStatement` as follows.

```
Statement statement = connection.createStatement();
```

Most, but not all, database drivers permit multiple concurrent `Statement` objects to be open on the same connection.

Execute a Query or Update

Once you have a `Statement` object, you can use it to send SQL queries by using the `executeQuery` method, which returns an object of type `ResultSet`. Here is an example.

```
String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);
```

The following list summarizes commonly used methods in the `Statement` class.

- **`executeQuery`.** Executes an SQL query and returns the data in a `ResultSet`. The `ResultSet` may be empty, but never `null`.
- **`executeUpdate`.** Used for `UPDATE`, `INSERT`, or `DELETE` commands. Returns the number of rows affected, which could be zero. Also provides support for Data Definition Language (DDL) commands, for example, `CREATE TABLE`, `DROP TABLE`, and `ALTER TABLE`.
- **`executeBatch`.** Executes a group of commands as a unit, returning an array with the update counts for each command. Use `addBatch` to add a command to the batch group. Note that vendors are not required to implement this method in their driver to be JDBC compliant.

- **`setQueryTimeout`.** Specifies the amount of time a driver waits for the result before throwing an `SQLException`.
- **`getMaxRows/setMaxRows`.** Determines the number of rows a `ResultSet` may contain. Excess rows are silently dropped. The default is zero for no limit.

In addition to using the methods described here to send arbitrary commands, you can use a `Statement` object to create parameterized queries by which values are supplied to a precompiled fixed-format query. See Section 17.4 (Using Prepared Statements) for details.

Process the Results

The simplest way to handle the results is to use the `next` method of `ResultSet` to move through the table a row at a time. Within a row, `ResultSet` provides various `getXxx` methods that take a column name or column index as an argument and return the result in a variety of different Java types. For instance, use `getInt` if the value should be an integer, `getString` for a `String`, and so on for most other data types. If you just want to display the results, you can use `getString` for most of the column types. However, if you use the version of `getXxx` that takes a column index (rather than a column name), note that columns are indexed starting at 1 (following the SQL convention), not at 0 as with arrays, vectors, and most other data structures in the Java programming language.

Core Warning

The first column in a `ResultSet` row has index 1, not 0.

Here is an example that prints the values of the first two columns and the first name and last name, for all rows of a `ResultSet`.

```
while(resultSet.next()) {
  System.out.println(resultSet.getString(1) + " " +
                     resultSet.getString(2) + " " +
                     resultSet.getString("firstname") + "   "
                     resultSet.getString("lastname"));
}
```

We suggest that when you access the columns of a `ResultSet`, you use the column name instead of the column index. That way, if the column structure of the table changes, the code interacting with the `ResultSet` will be less likely to fail.

Core Approach

Use the column name instead of the column index when accessing data in a `ResultSet`.

In JDBC 1.0, you can only move forward in the `ResultSet`; however, in JDBC 2.0, you can move forward (`next`) and backward (`previous`) in the `ResultSet` as well as move to a particular row (`relative`, `absolute`). In Volume 2 of this book, we present several custom tags that illustrate the JDBC 2.0 methods available in a `ResultSet`.

Be aware that neither JDBC 1.0 nor JDBC 2.0 provides a direct mechanism to determine the JDBC version of the driver. In JDBC 3.0, this problem is resolved by the addition of `getJDBCMajorVersion` and `getJDBCMinorVersion` methods to the `DatabaseMetaData` class. If the JDBC version is not clear from the vendor's documentation, you can write a short program to obtain a `ResultSet` and attempt a `previous` operation on the `ResultSet`. Since `resultSet.previous` is only available in JDBC 2.0 and later, a JDBC 1.0 driver would throw an exception at this point. See Section 18.4 (Testing Your Database Through a JDBC Connection) for an example program that performs a nonrigorous test to determine the JDBC version of your database driver.

The following list summarizes useful `ResultSet` methods.

- **`next`/`previous`.** Moves the cursor to the next (any JDBC version) or previous (JDBC version 2.0 or later) row in the `ResultSet`, respectively.
- **`relative`/`absolute`.** The `relative` method moves the cursor a relative number of rows, either positive (up) or negative (down). The `absolute` method moves the cursor to the given row number. If the absolute value is negative, the cursor is positioned relative to the end of the `ResultSet` (JDBC 2.0).
- **`get`*`Xxx`*.** Returns the value from the column specified by the column name or column index as an *`Xxx`* Java type (see `java.sql.Types`). Can return 0 or `null` if the value is an SQL `NULL`.
- **`wasNull`.** Checks whether the last `get`*`Xxx`* read was an SQL `NULL`. This check is important if the column type is a primitive (`int`, `float`, etc.) and the value in the database is 0. A zero value would be indistinguishable from a database value of `NULL`, which is also returned as a 0. If the column type is an object (`String`, `Date`, etc.), you can simply compare the return value to `null`.
- **`findColumn`.** Returns the index in the `ResultSet` corresponding to the specified column name.

- **`getRow`.** Returns the current row number, with the first row starting at 1 (JDBC 2.0).
- **`getMetaData`.** Returns a `ResultSetMetaData` object describing the `ResultSet`. `ResultSetMetaData` gives the number of columns and the column names.

The `getMetaData` method is particularly useful. Given only a `ResultSet`, you have to know the name, number, and type of the columns to be able to process the table properly. For most fixed-format queries, this is a reasonable expectation. For ad hoc queries, however, it is useful to be able to dynamically discover high-level information about the result. That is the role of the `ResultSetMetaData` class: it lets you determine the number, names, and types of the columns in the `ResultSet`.

Useful `ResultSetMetaData` methods are described below.

- **`getColumnCount`.** Returns the number of columns in the `ResultSet`.
- **`getColumnName`.** Returns the database name of a column (indexed starting at 1).
- **`getColumnType`.** Returns the SQL type, to compare with entries in `java.sql.Types`.
- **`isReadOnly`.** Indicates whether the entry is a read-only value.
- **`isSearchable`.** Indicates whether the column can be used in a `WHERE` clause.
- **`isNullable`.** Indicates whether storing `NULL` is legal for the column.

`ResultSetMetaData` does *not* include information about the number of rows; however, if your driver complies with JDBC 2.0, you can call `last` on the `ResultSet` to move the cursor to the last row and then call `getRow` to retrieve the current row number. In JDBC 1.0, the only way to determine the number of rows is to repeatedly call `next` on the `ResultSet` until it returns `false`.

Core Note

`ResultSet` and `ResultSetMetaData` do not directly provide a method to return the number of rows returned from a query. However, in JDBC 2.0, you can position the cursor at the last row in the `ResultSet` by calling `last`, and then obtain the current row number by calling `getRow`.

Close the Connection

To close the connection explicitly, you would do:

```
connection.close();
```

Closing the connection also closes the corresponding `Statement` and `ResultSet` objects.

You should postpone closing the connection if you expect to perform additional database operations, since the overhead of opening a connection is usually large. In fact, reusing existing connections is such an important optimization that the JDBC 2.0 API defines a `ConnectionPoolDataSource` interface for obtaining pooled connections. Pooled connections are discussed in Volume 2 of this book.

17.2 Basic JDBC Examples

In this section, we present two simple JDBC examples that connect to the Microsoft Access Northwind database (shown in Figure 17–2) and perform a simple query. The Northwind database is included in the samples section of Microsoft Office. To configure the Northwind database for access from JDBC, see Section 18.1.

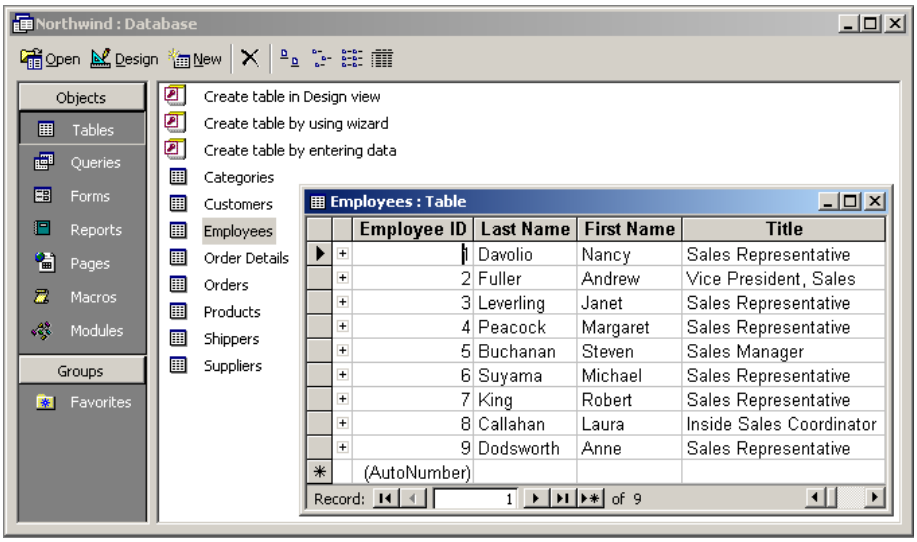

Figure 17–2 Microsoft Access Northwind sample database showing the first four columns of the Employees table. See Section 18.1 for information on using this database.

Northwind is a good database for testing and experimentation since it is already installed on many systems and since the JDBC-ODBC bridge for connecting to Microsoft Access is already bundled in the JDK. However, Microsoft Access is not intended for serious online databases. For production purposes, a higher-performance option like MySQL (see Section 18.2), Oracle9i (see Section 18.3), Microsoft SQL Server, Sybase, or DB2 is far better.

The first example, Listing 17.1, presents a standalone class called `NorthwindTest` that follows the seven steps outlined in the previous section to display the results of querying the `Employee` table.

The results for the `NorthwindTest` are shown in Listing 17.2. Since `NorthwindTest` is in the `coreservlets` package, it resides in a subdirectory called coreservlets. Before compiling the file, set the `CLASSPATH` to include the directory *containing* the coreservlets directory. See Section 2.7 (Set Up Your Development Environment) for details. With this setup, simply compile the program by running `javac NorthwindTest.java` from within the coreservlets subdirectory (or by selecting "build" or "compile" in your IDE). To run `NorthwindTest`, you need to refer to the full package name with `java coreservlets.NorthwindTest`.

The second example, Listing 17.3 (`NorthwindServlet`), connects to the database from a servlet and presents the query results as an HTML table. Both Listing 17.1 and Listing 17.3 use the JDBC-ODBC bridge driver, `sun.jdbc.odbc.JdbcOdbcDriver`, included with the JDK.

Listing 17.1 NorthwindTest.java

```
package coreservlets;

import java.sql.*;

/** A JDBC example that connects to the MicroSoft Access sample
 *  Northwind database, issues a simple SQL query to the
 *  employee table, and prints the results.
 */

public class NorthwindTest {
  public static void main(String[] args) {
    String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    String url = "jdbc:odbc:Northwind";
    String username = ""; // No username/password required
    String password = ""; // for desktop access to MS Access.
    showEmployeeTable(driver, url, username, password);
  }

  /** Query the employee table and print the first and
   *  last names.
   */
```

Listing 17.1 NorthwindTest.java *(continued)*

```
  public static void showEmployeeTable(String driver,
                                       String url,
                                       String username,
                                       String password) {
    try {
      // Load database driver if it's not already loaded.
      Class.forName(driver);
      // Establish network connection to database.
      Connection connection =
        DriverManager.getConnection(url, username, password);
      System.out.println("Employees\n" + "==========");
      // Create a statement for executing queries.
      Statement statement = connection.createStatement();
      String query =
        "SELECT firstname, lastname FROM employees";
      // Send query to database and store results.
      ResultSet resultSet = statement.executeQuery(query);
      // Print results.
      while(resultSet.next()) {
        System.out.print(resultSet.getString("firstname") + " ");
        System.out.println(resultSet.getString("lastname"));
      }
      connection.close();
    } catch(ClassNotFoundException cnfe) {
      System.err.println("Error loading driver: " + cnfe);
    } catch(SQLException sqle) {
      System.err.println("Error with connection: " + sqle);
    }
  }
}
```

Listing 17.2 NorthwindTest Result

```
Prompt> java coreservlets.NorthwindTest

Employees
==========
Nancy Davolio
Andrew Fuller
Janet Leverling
Margaret Peacock
Steven Buchanan
Michael Suyama
Robert King
Laura Callahan
Anne Dodsworth
```

For the second example, `NorthwindServlet` (Listing 17.3), the information for performing the query is taken from an HTML form, NorthwindForm.html, shown in Listing 17.4. Here, you can enter the query into the form text area before submitting the form to the servlet. The servlet reads the driver, URL, username, password, and the query from the request parameters and generates an HTML table based on the query results. The servlet also demonstrates the use of `DatabaseMetaData` to look up the product name and product version of the database. The HTML form is shown in Figure 17–3; Figure 17–4 shows the result of submitting the form. For this example, the HTML form and servlet are located in the Web application named `jdbc`. For more information on creating and using Web applications, see Section 2.11.

Listing 17.3 NorthwindServlet.java

```
package coreservlets;

import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** A simple servlet that connects to a database and
 *  presents the results from the query in an HTML
 *  table. The driver, URL, username, password,
 *  and query are taken from form input parameters.
 */

public class NorthwindServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\"\n";
    String title = "Northwind Results";
    out.print(docType +
              "<HTML>\n" +
              "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
              "<BODY BGCOLOR=\"#FDF5E6\"><CENTER>\n" +
              "<H1>Database Results</H1>\n");
    String driver = request.getParameter("driver");
    String url = request.getParameter("url");
    String username = request.getParameter("username");
```

Listing 17.3 NorthwindServlet.java *(continued)*

```
  String password = request.getParameter("password");
  String query = request.getParameter("query");
  showTable(driver, url, username, password, query, out);
  out.println("</CENTER></BODY></HTML>");
}

public void showTable(String driver, String url,
                      String username, String password,
                      String query, PrintWriter out) {
  try {
    // Load database driver if it's not already loaded.
    Class.forName(driver);
    // Establish network connection to database.
    Connection connection =
      DriverManager.getConnection(url, username, password);
    // Look up info about the database as a whole.
    DatabaseMetaData dbMetaData = connection.getMetaData();
    out.println("<UL>");
    String productName =
      dbMetaData.getDatabaseProductName();
    String productVersion =
      dbMetaData.getDatabaseProductVersion();
    out.println("  <LI><B>Database:</B> " + productName +
                "  <LI><B>Version:</B> " + productVersion +
                "</UL>");
    Statement statement = connection.createStatement();
    // Send query to database and store results.
    ResultSet resultSet = statement.executeQuery(query);
    // Print results.
    out.println("<TABLE BORDER=1>");
    ResultSetMetaData resultSetMetaData =
      resultSet.getMetaData();
    int columnCount = resultSetMetaData.getColumnCount();
    out.println("<TR>");
    // Column index starts at 1 (a la SQL), not 0 (a la Java).
    for(int i=1; i <= columnCount; i++) {
      out.print("<TH>" + resultSetMetaData.getColumnName(i));
    }
    out.println();
    // Step through each row in the result set.
    while(resultSet.next()) {
      out.println("<TR>");
```

Listing 17.3 NorthwindServlet.java *(continued)*

```
        // Step across the row, retrieving the data in each
        // column cell as a String.
        for(int i=1; i <= columnCount; i++) {
          out.print("<TD>" + resultSet.getString(i));
        }
        out.println();
      }
      out.println("</TABLE>");
      connection.close();
    } catch(ClassNotFoundException cnfe) {
      System.err.println("Error loading driver: " + cnfe);
    } catch(SQLException sqle) {
      System.err.println("Error connecting: " + sqle);
    } catch(Exception ex) {
      System.err.println("Error with input: " + ex);
    }
  }
}
```

Listing 17.4 NorthwindForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Simple Query Form</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H2>Query Input:</H2>
<FORM ACTION="/jdbc/servlet/coreservlets.NorthwindServlet"
      METHOD="POST">
<TABLE>
  <TR><TD>Driver:
      <TD><INPUT TYPE="TEXT" NAME="driver"
                 VALUE="sun.jdbc.odbc.JdbcOdbcDriver" SIZE="35">
  <TR><TD>URL:
      <TD><INPUT TYPE="TEXT" NAME="url"
                 VALUE="jdbc:odbc:Northwind" SIZE="35">
  <TR><TD>Username:
      <TD><INPUT TYPE="TEXT" NAME="username">
```

Listing 17.4 NorthwindForm.html *(continued)*

```
  <TR><TD>Password:
      <TD><INPUT TYPE="PASSWORD" NAME="password">
  <TR><TD VALIGN="TOP">Query:
      <TD><TEXTAREA ROWS="5" COLS="35" NAME="query"></TEXTAREA>
  <TR><TD COLSPAN="2" ALIGN="CENTER"><INPUT TYPE="SUBMIT">
</TABLE>
</FORM>
</BODY></HTML>
```

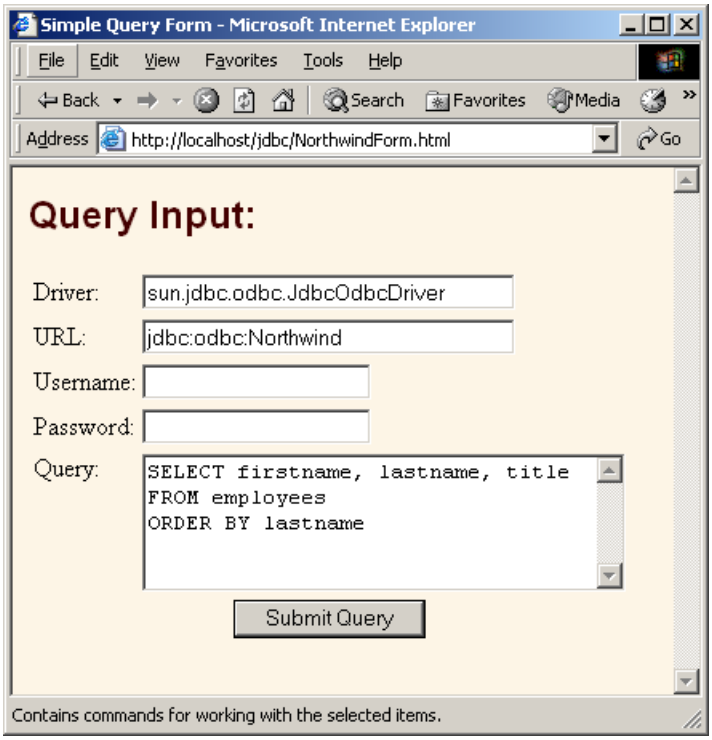

Figure 17–3 NorthwindForm.html: front end to servlet that queries the Northwind database.

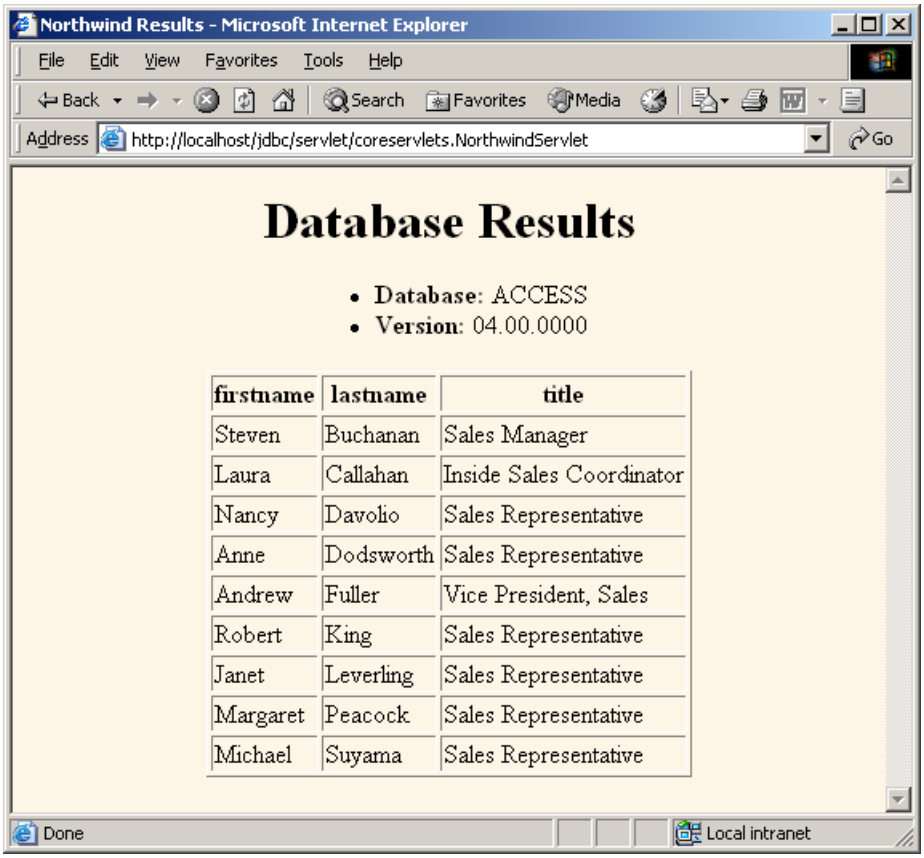

Figure 17–4 Result of querying the Northwind database.

In the preceding example, the HTML table was generated from the query results within a servlet. In Volume 2 of this book, we present various custom tags to generate the HTML table from the query results in the JSP page itself. Furthermore, if your development model favors JSP, the JSP Standard Tag Library (JSTL) provides an `sql:query` action to query a database and store the query result in a scoped variable for processing on the JSP page. JSTL is also covered in Volume 2 of this book.

17.3 Simplifying Database Access with JDBC Utilities

In this section, we present a couple of helper classes that are used throughout this chapter to simplify coding. These classes provide basic functionality for loading drivers and making database connections.

For example, the `DriverUtilities` class (Listing 17.5) simplifies the building of a URL to connect to a database. To build a URL for MySQL, which is in the form

```
String url = "jdbc:mysql://host:3306/dbname";
```

you first need to load the vendor data by calling `loadDrivers`. Then, call `makeURL` to build the URL, as in

```
DriverUtilities.loadDrivers();
String url =
  DriverUtilities.makeURL(host, dbname, DriverUtilities.MYSQL);
```

where the host, database name, and vendor are dynamically specified as arguments. In this manner, the database URL does not need to be hard-coded in the examples throughout this chapter. More importantly, you can simply add information about your database to the `loadDrivers` method in `DriverUtilities` (and a constant to refer to your driver, if desired). Afterwards, the examples throughout this chapter should work for your environment.

As another example, the `ConnectionInfoBean` class (Listing 17.9) provides a utility method, `getConnection`, for obtaining a `Connection` to a database. Thus, to obtain a connection to the database, replace

```
Connection connection = null;
try {
  Class.forName(driver);
  connection = DriverManager.getConnection(url, username,
                                           password);
} catch(ClassNotFoundException cnfe) {
  System.err.println("Error loading driver: " + cnfe);
} catch(SQLException sqle) {
  System.err.println("Error connecting: " + sqle);
}
```

with

```
Connection connection =
  ConnectionInfoBean.getConnection(driver, url,
                                   username, password);
```

If an `SQLException` occurs while the connection is being acquired, `null` is returned.

We define four utility classes in this section.

1. **`DriverUtilities`**
   This class, shown in Listing 17.5, loads explicitly coded driver information about various database vendors. It then provides methods for obtaining the driver class for a vendor (`getDriver`) and creating a URL (`makeURL`), given the host, database name, and vendor. We provide driver information for Microsoft Access, MySQL, and Oracle databases, but you can easily update the class for your environment.
2. **`DriverUtilities2`**
   This class, shown in Listing 17.6, extends `DriverUtilities` (Listing 17.5) and overrides `loadDrivers` to obtain the driver information from an XML file. A representative XML file is shown in drivers.xml, Listing 17.7.
3. **`DriverInfoBean`**
   The `DriverInfoBean` class, shown in Listing 17.8, encapsulates driver information for a specific vendor (used by `DriverUtilities`, Listing 17.5). The bean contains a keyword (vendor name), a brief description of the driver, the driver classname, and a URL for connecting to a database.
4. **`ConnectionInfoBean`**
   This class, shown in Listing 17.9, encapsulates information for connection to a *particular* database. The bean encapsulates a name for the connection, a brief description of the connection, the driver class, the URL to connect to the database, the username, and the password. In addition, the bean provides a `getConnection` method to directly obtain a `Connection` to a database.

Listing 17.5 DriverUtilities.java

```
package coreservlets;

import java.io.*;
import java.sql.*;
import java.util.*;
import coreservlets.beans.*;
```

Listing 17.5 DriverUtilities.java *(continued)*

```
/** Some simple utilities for building JDBC connections to
 *  databases from different vendors. The drivers loaded are
 *  hard-coded and specific to our local setup. You can
 *  either modify the loadDrivers method and recompile or
 *  use <CODE>DriverUtilities2</CODE> to load the driver
 *  information for each vendor from an XML file.
 */

public class DriverUtilities {
  public static final String MSACCESS = "MSACCESS";
  public static final String MYSQL = "MYSQL";
  public static final String ORACLE = "ORACLE";

  // Add constant to refer to your database here ...

  protected static Map driverMap = new HashMap();

  /** Load vendor driver information. Here we have hard-coded
   *  driver information specific to our local setup.
   *  Modify the values according to your setup.
   *  Alternatively, you can use <CODE>DriverUtilities2</CODE>
   *  to load driver information from an XML file.
   *  <P>
   *  Each vendor is represented by a
   *  <CODE>DriverInfoBean</CODE> that defines a vendor
   *  name (keyword), description, driver class, and URL. The
   *  bean is stored in a driver map; the vendor name is
   *  used as the keyword to retrieve the information.
   *  <P>
   *  The url variable should contain the placeholders
   *  [$host] and [$dbName] to substitute for the <I>host</I>
   *  and <I>database name</I> in <CODE>makeURL</CODE>.
   */

  public static void loadDrivers() {
    String vendor, description, driverClass, url;
    DriverInfoBean info = null;

    // MSAccess
    vendor = MSACCESS;
    description = "MS Access 4.0";
    driverClass = "sun.jdbc.odbc.JdbcOdbcDriver";
    url = "jdbc:odbc:[$dbName]";
    info = new DriverInfoBean(vendor, description,
                              driverClass, url);
    addDriverInfoBean(info);
```

Listing 17.5 DriverUtilities.java *(continued)*

```
  // MySQL
  vendor = MYSQL;
  description = "MySQL Connector/J 3.0";
  driverClass = "com.mysql.jdbc.Driver";
  url = "jdbc:mysql://[$host]:3306/[$dbName]";
  info = new DriverInfoBean(vendor, description,
                            driverClass, url);
  addDriverInfoBean(info);

  // Oracle
  vendor = ORACLE;
  description = "Oracle9i Database";
  driverClass = "oracle.jdbc.driver.OracleDriver";
  url = "jdbc:oracle:thin:@[$host]:1521:[$dbName]";
  info = new DriverInfoBean(vendor, description,
                            driverClass, url);
  addDriverInfoBean(info);

  // Add info on your database here...
}

/** Add information (<CODE>DriverInfoBean</CODE>) about new
 *  vendor to the map of available drivers.
 */

public static void addDriverInfoBean(DriverInfoBean info) {
  driverMap.put(info.getVendor().toUpperCase(), info);
}

/** Determine if vendor is represented in the loaded
 *  driver information.
 */

public static boolean isValidVendor(String vendor) {
  DriverInfoBean info =
    (DriverInfoBean)driverMap.get(vendor.toUpperCase());
  return(info != null);
}

/** Build a URL in the format needed by the
 *  database drivers. In building of the final URL, the
 *  keywords [$host] and [$dbName] in the URL
 *  (looked up from the vendor's <CODE>DriverInfoBean</CODE>)
 *  are appropriately substituted by the host and dbName
 *  method arguments.
 */
```

Listing 17.5 DriverUtilities.java *(continued)*

```
  public static String makeURL(String host, String dbName,
                               String vendor) {
    DriverInfoBean info =
      (DriverInfoBean)driverMap.get(vendor.toUpperCase());
    if (info == null) {
      return(null);
    }
    StringBuffer url = new StringBuffer(info.getURL());
    DriverUtilities.replace(url, "[$host]", host);
    DriverUtilities.replace(url, "[$dbName]", dbName);
    return(url.toString());
  }

  /** Get the fully qualified name of a driver. */

  public static String getDriver(String vendor) {
    DriverInfoBean info =
      (DriverInfoBean)driverMap.get(vendor.toUpperCase());
    if (info == null) {
      return(null);
    } else {
      return(info.getDriverClass());
    }
  }

  /** Perform a string substitution, where the first "match"
   *  is replaced by the new "value".
   */

  private static void replace(StringBuffer buffer,
                              String match, String value) {
    int index = buffer.toString().indexOf(match);
    if (index > 0) {
      buffer.replace(index, index + match.length(), value);
    }
  }
}
```

In `DriverUtilities`, driver information for each vendor (Microsoft Access, MySQL, and Oracle9i) is explicitly coded into the program. If you are using a different database, you will need to modify `DriverUtilities` to include your driver information and then recompile the code. Since this approach may not be convenient, we include a second program, `DriverUtilities2` in Listing 17.6, that reads

the driver information from an XML file. Then, to add a new database vendor to your program, you simply edit the XML file. An example XML file, drivers.xml, is given in Listing 17.7

When using `DriverUtilites2` in a command-line application, place the driver file, drivers.xml, in the working directory from which you started the application. Afterwards, call `loadDrivers` with the complete filename (including path).

For a Web application, we recommend placing drivers.xml in the WEB-INF directory. You may want to specify the filename as a context initialization parameter in web.xml (for details, see the chapter on web.xml in Volume 2 of this book). Also, remember that from the servlet context you can use `getRealPath` to determine the physical path to a file relative to the Web application directory, as shown in the following code fragment.

```
ServletContext context = getServletContext();
String path = context.getRealPath("/WEB-INF/drivers.xml");
```

JDK 1.4 includes all the necessary classes to parse the XML document, drivers.xml. If you are using JDK 1.3 or earlier, you will need to download and install a SAX and DOM parser. Xerces-J by Apache is an excellent parser and is available at http://xml.apache.org/xerces2-j/. Most Web application servers are already bundled with an XML parser, so you may not need to download Xerces-J. Check the vendor's documentation to determine where the parser files are located and include them in your `CLASSPATH` for compiling your application. For example, Tomcat 4.x includes the parser JAR files (xercesImpl.jar and xmlParserAPI.jar) in the *install_dir*/common/endorsed directory.

Note that if you are using servlets 2.4 (JSP 2.0) in a fully J2EE-1.4-compatible server, you are guaranteed to have JDK 1.4 or later.

Listing 17.6 DriverUtilities2.java

```
package coreservlets;

import java.io.*;
import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import coreservlets.beans.*;

/** Extends <CODE>DriverUtilities</CODE> to support the
 *  loading of driver information for different database vendors
 *  from an XML file (default is drivers.xml). Both DOM and
 *  JAXP are used to read the XML file.  The format for the
 *  XML file is:
```

Listing 17.6 DriverUtilities2.java *(continued)*

```
 *  <P>
 *  <PRE>
 *    &lt;drivers&gt;
 *      &lt;driver&gt;
 *        &lt;vendor&gt;ORACLE&lt;/vendor&gt;
 *        &lt;description&gt;Oracle&lt;/description&gt;
 *        &lt;driver-class&gt;
 *          oracle.jdbc.driver.OracleDriver
 *        &lt;/driver-class&gt;
 *        &lt;url&gt;
 *          jdbc:oracle:thin:@[$host]:1521:[$dbName]
 *        &lt;/url&gt;
 *      &lt;/driver&gt;
 *      ...
 *    &lt;drivers&gt;
 *  </PRE>
 *  <P>
 *  The url element should contain the placeholders
 *  [$host] and [$dbName] to substitute for the host and
 *  database name in makeURL.
 */

public class DriverUtilities2 extends DriverUtilities {
  public static final String DEFAULT_FILE = "drivers.xml";

  /** Load driver information from default XML file,
   *  drivers.xml.
   */

  public static void loadDrivers() {
    DriverUtilities2.loadDrivers(DEFAULT_FILE);
  }

  /** Load driver information from specified XML file. Each
   *  vendor is represented by a <CODE>DriverInfoBean</CODE>
   *  object and stored in the map, with the vendor name as
   *  the key. Use this method if you need to load a
   *  driver file other than the default, drivers.xml.
   */

  public static void loadDrivers(String filename) {
    File file = new File(filename);
    try {
      InputStream in = new FileInputStream(file);
      DocumentBuilderFactory builderFactory =
        DocumentBuilderFactory.newInstance();
```

Listing 17.6 DriverUtilities2.java *(continued)*

```
      DocumentBuilder builder =
        builderFactory.newDocumentBuilder();
      Document document = builder.parse(in);
      document.getDocumentElement().normalize();
      Element rootElement = document.getDocumentElement();
      NodeList driverElements =
        rootElement.getElementsByTagName("driver");
      // Build DriverInfoBean for each vendor
      for(int i=0; i<driverElements.getLength(); i++) {
        Node node = driverElements.item(i);
        DriverInfoBean info =
          DriverUtilities2.createDriverInfoBean(node);
        if (info != null) {
          addDriverInfoBean(info);
        }
      }
    } catch(FileNotFoundException fnfe) {
      System.err.println("Can't find " + filename);
    } catch(IOException ioe) {
      System.err.println("Problem reading file: " + ioe);
    } catch(ParserConfigurationException pce) {
      System.err.println("Can't create DocumentBuilder");
    } catch(SAXException se) {
      System.err.println("Problem parsing document: " + se);
    }
  }

  /** Build a DriverInfoBean object from an XML DOM node
   *  representing a vendor driver in the format:
   *  <P>
   *  <PRE>
   *    &lt;driver&gt;
   *      &lt;vendor&gt;ORACLE&lt;/vendor&gt;
   *      &lt;description&gt;Oracle&lt;/description&gt;
   *      &lt;driver-class&gt;
   *         oracle.jdbc.driver.OracleDriver
   *      &lt;/driver-class&gt;
   *      &lt;url&gt;
   *        jdbc:oracle:thin:@[$host]:1521:[$dbName]
   *      &lt;/url&gt;
   *    &lt;/driver&gt;
   *  </PRE>
   */
```

Listing 17.6 DriverUtilities2.java *(continued)*

```
  public static DriverInfoBean createDriverInfoBean(Node node) {
    Map map = new HashMap();
    NodeList children = node.getChildNodes();
    for(int i=0; i<children.getLength(); i++) {
      Node child = children.item(i);
      String nodeName = child.getNodeName();
      if (child instanceof Element) {
        Node textNode = child.getChildNodes().item(0);
        if (textNode != null) {
          map.put(nodeName, textNode.getNodeValue());
        }
      }
    }
    return(new DriverInfoBean((String)map.get("vendor"),
                              (String)map.get("description"),
                              (String)map.get("driver-class"),
                              (String)map.get("url")));
  }
}
```

Listing 17.7 drivers.xml

```
<?xml version="1.0"?>
<!--
Used by DriverUtilities2. Here you configure information
about your database server in XML. To add a driver, include
a vendor keyword, description, driver-class, and URL.
For general use, the host and database name should not
be included in the URL; a special notation is required
for the host and database name. Use [$host] as a
placeholder for the host server and [$dbName] as a placeholder
for the database name. Specify the actual host and database name
when making a call to makeUrl (DriverUtilities). Then, the
appropriate strings will be substituted for [$host]
and [$dbName] before the URL is returned.
-->
<drivers>
  <driver>
    <vendor>MSACCESS</vendor>
    <description>MS Access</description>
    <driver-class>sun.jdbc.odbc.JdbcOdbcDriver</driver-class>
    <url>jdbc:odbc:[$dbName]</url>
  </driver>
```

Listing 17.7 drivers.xml *(continued)*

```
  <driver>
    <vendor>MYSQL</vendor>
    <description>MySQL Connector/J 3.0</description>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <url>jdbc:mysql://[$host]:3306/[$dbName]</url>
  </driver>
  <driver>
    <vendor>ORACLE</vendor>
    <description>Oracle</description>
    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
    <url>jdbc:oracle:thin:@[$host]:1521:[$dbName]</url>
  </driver>
</drivers>
```

Listing 17.8 DriverInfoBean.java

```
package coreservlets.beans;

/** Driver information for a vendor.  Defines the vendor
 *  keyword, description, driver class, and URL construct for
 *  connecting to a database.
 */

public class DriverInfoBean {
  private String vendor;
  private String description;
  private String driverClass;
  private String url;

public class DriverInfoBean {
  private String vendor;
  private String description;
  private String driverClass;
  private String url;

  public DriverInfoBean(String vendor,
                        String description,
                        String driverClass,
                        String url) {
    this.vendor = vendor;
    this.description = description;
    this.driverClass = driverClass;
    this.url = url;
  }
```

Listing 17.8 DriverInfoBean.java *(continued)*

```
  public String getVendor() {
    return(vendor);
  }

  public String getDescription() {
    return(description);
  }

  public String getDriverClass() {
    return(driverClass);
  }

  public String getURL() {
    return(url);
  }
}
```

Listing 17.9 ConnectionInfoBean.java

```
package coreservlets.beans;

import java.sql.*;

/** Stores information to create a JDBC connection to
 *  a database. Information includes:
 *  <UL>
 *    <LI>connection name
 *    <LI>description of the connection
 *    <LI>driver classname
 *    <LI>URL to connect to the host
 *    <LI>username
 *    <LI>password
 *  </UL>
 */

public class ConnectionInfoBean {
  private String connectionName;
  private String description;
  private String driver;
  private String url;
  private String username;
  private String password;
```

Listing 17.9 ConnectionInfoBean.java *(continued)*

```
public ConnectionInfoBean() { }

public ConnectionInfoBean(String connectionName,
                          String description,
                          String driver,
                          String url,
                          String username,
                          String password) {
  setConnectionName(connectionName);
  setDescription(description);
  setDriver(driver);
  setURL(url);
  setUsername(username);
  setPassword(password);
}

public void setConnectionName(String connectionName) {
  this.connectionName = connectionName;
}

public String getConnectionName() {
  return(connectionName);
}

public void setDescription(String description) {
  this.description = description;
}

public String getDescription() {
  return(description);
}

public void setDriver(String driver) {
  this.driver = driver;
}

public String getDriver() {
  return(driver);
}

public void setURL(String url) {
  this.url = url;
}

public String getURL() {
  return(url);
}
```

Listing 17.9 ConnectionInfoBean.java *(continued)*

```
  public void setUsername(String username) {
    this.username = username;
  }

  public String getUsername() {
    return(username);
  }

  public void setPassword(String password) {
    this.password = password;
  }

  public String getPassword() {
    return(password);
  }

  public Connection getConnection() {
    return(getConnection(driver, url, username, password));

  }

  /** Create a JDBC connection or return null if a
   *  problem occurs.
   */

  public static Connection getConnection(String driver,
                                         String url,
                                         String username,
                                         String password) {
    try {
      Class.forName(driver);
      Connection connection =
        DriverManager.getConnection(url, username,
                                    password);
      return(connection);
    } catch(ClassNotFoundException cnfe) {
      System.err.println("Error loading driver: " + cnfe);
      return(null);
    } catch(SQLException sqle) {
      System.err.println("Error connecting: " + sqle);
      return(null);
    }
  }
}
```

17.4 Using Prepared Statements

If you are going to execute similar SQL statements multiple times, using parameterized (or "prepared") statements can be more efficient than executing a raw query each time. The idea is to create a parameterized statement in a standard form that is sent to the database for compilation before actually being used. You use a question mark to indicate the places where a value will be substituted into the statement. Each time you use the prepared statement, you simply replace the marked parameters, using a `setXxx` call corresponding to the entry you want to set (using 1-based indexing) and the type of the parameter (e.g., `setInt`, `setString`). You then use `executeQuery` (if you want a `ResultSet` back) or `execute`/`executeUpdate` to modify table data, as with normal statements.

For instance, in Section 18.5, we create a `music` table summarizing the price and availability of concerto recordings for various classical composers. Suppose, for an upcoming sale, you want to change the price of all the recordings in the `music` table. You might do something like the following.

```
Connection connection =
  DriverManager.getConnection(url, username, password);
String template =
  "UPDATE music SET price = ? WHERE id = ?";
PreparedStatement statement =
  connection.prepareStatement(template);
float[] newPrices = getNewPrices();
int[] recordingIDs = getIDs();
for(int i=0; i<recordingIDs.length; i++) {
  statement.setFloat(1, newPrices[i]);  // Price
  statement.setInt(2, recordingIDs[i]); // ID
  statement.execute();
}
```

The performance advantages of prepared statements can vary significantly, depending on how well the server supports precompiled queries and how efficiently the driver handles raw queries. For example, Listing 17.10 presents a class that sends 100 different queries to a database, using prepared statements, then repeats the same 100 queries, using regular statements. On one hand, with a PC and fast LAN connection (100 Mbps) to an Oracle9i database, prepared statements took only about 62 percent of the time required by raw queries, averaging 0.61 seconds for the 100 queries as compared with an average of 0.99 seconds for the regular statements (average of 5 runs). On the other hand, with MySQL (Connector/J 3.0) the prepared statement times were nearly identical to the raw queries with a fast LAN connection, with only about an 8 percent reduction in query time. To get performance numbers for your setup, download DriverUtilities.java from http://www.coreservlets.com/, add information about your drivers

to it, then run the `PreparedStatements` program yourself. To create the `music` table, see Section 18.5.

Be cautious though: a prepared statement does not always execute faster than an ordinary SQL statement. The performance improvement can depend on the particular SQL command you are executing. For a more detailed analysis of the performance for prepared statements in Oracle, see http://www.oreilly.com/catalog/jorajdbc/chapter/ch19.html.

However, performance is not the only advantage of a prepared statement. Security is another advantage. We recommend that you always use a prepared statement or stored procedure (see Section 17.5) to update database values when accepting input from a user through an HTML form. This approach is strongly recommended over the approach of building an SQL statement by concatenating strings from the user input values. Otherwise, a clever attacker could submit form values that look like portions of SQL statements, and once those were executed, the attacker could inappropriately access or modify the database. This security risk is often referred to as an SQL Injection Attack. In addition to removing the risk of a such an attack, a prepared statement will properly handle embedded quotes in strings and handle noncharacter data (e.g., sending a serialized object to a database).

Core Approach

To avoid an SQL Injection Attack when accepting data from an HTML form, use a prepared statement or stored procedure to update the database.

Listing 17.10 PreparedStatements.java

```
package coreservlets;

import java.sql.*;
import coreservlets.beans.*;

/** An example to test the timing differences resulting
 *  from repeated raw queries vs. repeated calls to
 *  prepared statements. These results will vary dramatically
 *  among database servers and drivers. With our setup
 *  and drivers, Oracle9i prepared statements took only 62% of
 *  the time that raw queries required, whereas MySQL
 *  prepared statements took nearly the same time as
 *  raw queries, with only an 8% improvement.
 */
```

Listing 17.10 PreparedStatements.java *(continued)*

```
public class PreparedStatements {
  public static void main(String[] args) {
    if (args.length < 5) {
      printUsage();
      return;
    }
    String vendor = args[4];
    // Use DriverUtilities2.loadDrivers() to load
    // the drivers from an XML file.
    DriverUtilities.loadDrivers();
    if (!DriverUtilities.isValidVendor(vendor)) {
      printUsage();
      return;
    }
    String driver = DriverUtilities.getDriver(vendor);
    String host = args[0];
    String dbName = args[1];
    String url =
      DriverUtilities.makeURL(host, dbName, vendor);
    String username = args[2];
    String password = args[3];
    // Use "print" only to confirm it works properly,
    // not when getting timing results.
    boolean print = false;
    if ((args.length > 5) && (args[5].equals("print"))) {
      print = true;
    }
    Connection connection =
      ConnectionInfoBean.getConnection(driver, url,
                                       username, password);
    if (connection != null) {
      doPreparedStatements(connection, print);
      doRawQueries(connection, print);
    }
    try {
      connection.close();
    } catch(SQLException sqle) {
      System.err.println("Problem closing connection: " + sqle);
    }
  }

  private static void doPreparedStatements(Connection conn,
                                           boolean print) {
    try {
      String queryFormat =
        "SELECT id FROM music WHERE price < ?";
```

Listing 17.10 PreparedStatements.java *(continued)*

```
      PreparedStatement statement =
        conn.prepareStatement(queryFormat);
      long startTime = System.currentTimeMillis();
      for(int i=0; i<100; i++) {
        statement.setFloat(1, i/4);
        ResultSet results = statement.executeQuery();
        if (print) {
          showResults(results);
        }
      }
      long stopTime = System.currentTimeMillis();
      double elapsedTime = (stopTime - startTime)/1000.0;
      System.out.println("Executing prepared statement " +
                         "100 times took " +
                         elapsedTime + " seconds.");
    } catch(SQLException sqle) {
      System.err.println("Error executing statement: " + sqle);
    }
  }

  public static void doRawQueries(Connection conn,
                                  boolean print) {
    try {
      String queryFormat =
        "SELECT id FROM music WHERE price < ";
      Statement statement = conn.createStatement();
      long startTime = System.currentTimeMillis();
      for(int i=0; i<100; i++) {
        ResultSet results =
          statement.executeQuery(queryFormat + i/4);
        if (print) {
          showResults(results);
        }
      }
      long stopTime = System.currentTimeMillis();
      double elapsedTime = (stopTime - startTime)/1000.0;
      System.out.println("Executing raw query " +
                         "100 times took " +
                         elapsedTime + " seconds.");
    } catch(SQLException sqle) {
      System.err.println("Error executing query: " + sqle);
    }
  }
```

Listing 17.10 PreparedStatements.java *(continued)*

```
  private static void showResults(ResultSet results)
      throws SQLException {
    while(results.next()) {
      System.out.print(results.getString(1) + " ");
    }
    System.out.println();
  }

  private static void printUsage() {
    System.out.println("Usage: PreparedStatements host " +
                       "dbName username password " +
                       "vendor [print].");
  }
}
```

The preceding example illustrates how to create a prepared statement and set parameters for the statement in a command-line program. For Web development, you may want to submit prepared statements to the database from a JSP page. If so, the JSP Standard Tag Library (JSTL—see Volume 2 of this book) provides an `sql:query` action to define a prepared statement for submission to the database and an `sql:param` action to specify parameter values for the prepared statement.

17.5 Creating Callable Statements

With a `CallableStatement`, you can execute a stored procedure or function in a database. For example, in an Oracle database, you can write a procedure or function in PL/SQL and store it in the database along with the tables. Then, you can create a connection to the database and execute the stored procedure or function through a `CallableStatement`.

A stored procedure has many advantages. For instance, syntax errors are caught at compile time instead of at runtime; the database procedure may run much faster than a regular SQL query; and the programmer only needs to know about the input and output parameters, not the table structure. In addition, coding of the stored procedure may be simpler in the database language than in the Java programming language because access to native database capabilities (sequences, triggers, multiple cursors) is possible.

One disadvantage of a stored procedure is that you may need to learn a new database-specific language (note, however, that Oracle8i Database and later support stored procedures written in the Java programming language). A second disadvantage

is that the business logic of the stored procedure executes on the database server instead of on the client machine or Web server. The industry trend has been to move as much business logic as possible from the database and to place the business logic in JavaBeans components (or, on large systems, Enterprise JavaBeans components) executing on the Web server. The main motivation for this approach in a Web architecture is that the database access and network I/O are often the performance bottlenecks.

Calling a stored procedure in a database involves the six basic steps outlined below and then described in detail in the following subsections.

1. **Define the call to the database procedure.** As with a prepared statement, you use special syntax to define a call to a stored procedure. The procedure definition uses escape syntax, where the appropriate `?` defines input and output parameters.
2. **Prepare a `CallableStatement` for the procedure.** You obtain a `CallableStatement` from a `Connection` by calling `prepareCall`.
3. **Register the output parameter types.** Before executing the procedure, you must declare the type of each output parameter.
4. **Provide values for the input parameters.** Before executing the procedure, you must supply the input parameter values.
5. **Execute the stored procedure.** To execute the database stored procedure, call `execute` on the `CallableStatement`.
6. **Access the returned output parameters.** Call the corresponding `get`*`Xxx`* method, according to the output type.

Define the Call to the Database Procedure

Creating a `CallableStatement` is somewhat similar to creating a `PreparedStatement` (see Section 17.4, "Using Prepared Statements") in that special SQL escape syntax is used in which the appropriate `?` is replaced with a value before the statement is executed. The definition for a procedure takes four general forms.

- **Procedure with no parameters.**
  ```
  { call procedure_name }
  ```
- **Procedure with input parameters.**
  ```
  { call procedure_name(?, ?, ...) }
  ```
- **Procedure with an output parameter.**
  ```
  { ? call procedure_name }
  ```
- **Procedure with input and output parameters.**
  ```
  { ? = call procedure_name(?, ?, ...) }
  ```

In each of the four procedure forms, the *`procedure_name`* is the name of the stored procedure in the database. Also, be aware that a procedure can return more

than one output parameter and that the indexed parameter values begin with the output parameters. Thus, in the last procedure example above, the first *input* parameter is indexed by a value of 2 (not 1).

Core Note

If the procedure returns output parameters, then the index of the input parameters must account for the number of output parameters.

Prepare a CallableStatement for the Procedure

You obtain a `CallableStatement` from a `Connection` with the `prepareCall` method, as below.

```
String procedure = "{ ? = call procedure_name( ?, ? ) }";
CallableStatement statement =
  connection.prepareCall(procedure);
```

Register the Output Parameter Types

You must register the JDBC type of each output parameter, using `registerOutParameter`, as follows,

```
statement.registerOutParameter(n, type);
```

where *n* corresponds to the ordered output parameter (using 1-based indexing), and *type* corresponds to a constant defined in the `java.sql.Types` class (`Types.FLOAT`, `Types.DATE`, etc.).

Provide Values for the Input Parameters

Before executing the stored procedure, you replace the marked input parameters by using a `set`*Xxx* call corresponding to the entry you want to set and the type of parameter (e.g., `setInt`, `setString`). For example,

```
statement.setString(2, "name");
statement.setFloat(3, 26.0F);
```

sets the first input parameter (presuming one output parameter) to a `String`, and the second input parameter to a `float`. Remember that if the procedure has output parameters, the index of the input parameters starts from the first output parameter.

Execute the Stored Procedure

To execute the stored procedure, simply call `execute` on the `CallableStatement` object. For example:

```
statement.execute();
```

Access the Output Parameters

If the procedure returns output parameters, then after you call `execute`, you can access each corresponding output parameter by calling `getXxx`, where *Xxx* corresponds to the type of return parameter (`getDouble`, `getDate`, etc.). For example,

```
int value = statement.getInt(1);
```

returns the first output parameter as a primitive `int`.

Example

In Listing 17.11, the `CallableStatements` class demonstrates the execution of an Oracle stored procedure (technically, a function, since it returns a value) written for the `music` table (see Section 18.5 for setting up the `music` table). You can create the `discount` stored procedure in the database by invoking our `CallableStatements` class and specifying `create` on the command line. Doing so calls the `createStoredFunction` method, which submits the procedure (a long string) to the database as an SQL update. Alternatively, if you have Oracle SQL*Plus, you can load the procedure directly from `discount.sql`, Listing 17.12. See Section 18.5 for information on running the SQL script in SQL*Plus.

The stored procedure `discount` modifies the `price` entry in the `music` table. Specifically, the procedure accepts two input parameters, `composer_in` (the composer to select in the `music` table) and `discount_in` (the percent by which to discount the price). If the `discount_in` is outside the range 0.05 to 0.50, then a value of `-1` is returned; otherwise, the number of rows modified in the table is returned from the stored procedure.

Listing 17.11 CallableStatements.java

```
package coreservlets;

import java.sql.*;
import coreservlets.beans.*;
```

Listing 17.11 CallableStatements.java *(continued)*

```
/** An example that executes the Oracle stored procedure
 *  "discount". Specifically, the price of all compositions
 *  by Mozart in the "music" table are discounted by
 *  10 percent.
 *  <P>
 *  To create the stored procedure, specify a command-line
 *  argument of "create".
 */

public class CallableStatements {
  public static void main(String[] args) {
    if (args.length < 5) {
      printUsage();
      return;
    }
    String vendor = args[4];
    // Change to DriverUtilities2.loadDrivers() to force
    // loading of vendor drivers from default XML file.
    DriverUtilities.loadDrivers();
    if (!DriverUtilities.isValidVendor(vendor)) {
      printUsage();
      return;
    }
    String driver = DriverUtilities.getDriver(vendor);
    String host = args[0];
    String dbName = args[1];
    String url =
      DriverUtilities.makeURL(host, dbName, vendor);
    String username = args[2];
    String password = args[3];

    Connection connection =
      ConnectionInfoBean.getConnection(driver, url,
                                       username, password);
    if (connection == null) {
      return;
    }

    try {
      if ((args.length > 5) && (args[5].equals("create"))) {
        createStoredFunction(connection);
      }
      doCallableStatement(connection, "Mozart", 0.10F);
    } catch(SQLException sqle) {
      System.err.println("Problem with callable: " + sqle);
```

Listing 17.11 CallableStatements.java *(continued)*

```
    } finally {
      try {
        connection.close();
      } catch(SQLException sqle) {
        System.err.println("Error closing connection: " + sqle);
      }
    }
  }

  private static void doCallableStatement(Connection connection,
                                          String composer,
                                          float discount)
      throws SQLException {
    CallableStatement statement = null;
    try {
      connection.prepareCall("{ ? = call discount( ?, ? ) }");
      statement.setString(2, composer);
      statement.setFloat(3, discount);
      statement.registerOutParameter(1, Types.INTEGER);
      statement.execute();
      int rows = statement.getInt(1);
      System.out.println("Rows updated: " + rows);
    } catch(SQLException sqle) {
      System.err.println("Problem with callable: " + sqle);
    } finally {
      if (statement != null) {
        statement.close();
      }
    }
  }

  /** Create the Oracle PL/SQL stored procedure "discount".
   *  The procedure (technically, a PL/SQL function, since a
   *  value is returned), discounts the price for the specified
   *  composer in the "music" table.
   */

  private static void createStoredFunction(
                        Connection connection)
      throws SQLException {
    String sql = "CREATE OR REPLACE FUNCTION discount " +
                 "  (composer_in IN VARCHAR2, " +
                 "   discount_in IN NUMBER) " +
                 "RETURN NUMBER " +
                 "IS " +
```

Listing 17.11 CallableStatements.java *(continued)*

```
                "  min_discount CONSTANT NUMBER:= 0.05; " +
                "  max_discount CONSTANT NUMBER:= 0.50; " +
                "BEGIN " +
                "  IF discount_in BETWEEN min_discount " +
                "                 AND max_discount THEN " +
                "    UPDATE music " +
                "    SET price = price * (1.0 - discount_in) "+
                "    WHERE composer = composer_in; " +
                "    RETURN(SQL%ROWCOUNT); " +
                "  ELSE " +
                "    RETURN(-1); " +
                "  END IF; " +
                "END discount;";
    Statement statement = null;
    try {
      statement = connection.createStatement();
      statement.executeUpdate(sql);
    } catch(SQLException sqle) {
      System.err.println("Problem creating function: " + sqle);
    } finally {
      if (statement != null) {
        statement.close();
      }
    }
  }

  private static void printUsage() {
    System.out.println("Usage: CallableStatement host " +
                       "dbName username password " +
                       "vendor [create].");
  }
}
```

Listing 17.12 discount.sql (PL/SQL function for Oracle)

```
/* Discounts the price of all music by the specified
 * composer, composer_in. The music is discounted by the
 * percentage specified by discount_in.
 *
 * Returns the number of rows modified, or -1 if the discount
 * value is invalid.
 */
```

Listing 17.12 discount.sql (PL/SQL function for Oracle) *(continued)*

```
CREATE OR REPLACE FUNCTION discount
  (composer_in IN VARCHAR2, discount_in IN NUMBER)
RETURN NUMBER
IS
  min_discount CONSTANT NUMBER:= 0.05;
  max_discount CONSTANT NUMBER:= 0.50;
BEGIN
  IF discount_in BETWEEN min_discount AND max_discount THEN
    UPDATE music
    SET price = price * (1.0 - discount_in)
    WHERE composer = composer_in;
    RETURN(SQL%ROWCOUNT);
  ELSE
    RETURN(-1);
  END IF;
END discount;
```

17.6 Using Database Transactions

When a database is updated, by default the changes are permanently written (or *committed*) to the database. However, this default behavior can be programmatically turned off. If autocommitting is turned off and a problem occurs with the updates, then each change to the database can be backed out (or rolled back to the original values). If the updates execute successfully, then the changes can later be permanently committed to the database. This approach is known as *transaction management*.

Transaction management helps to ensure the integrity of your database tables. For example, suppose you are transferring funds from a savings account to a checking account. If you first withdraw from the savings account and then deposit into the checking account, what happens if there is an error after the withdrawal but before the deposit? The customer's accounts will have too little money, and the banking regulators will levy stiff fines. On the other hand, what if you first deposit into the checking account and then withdraw from the savings account, and there is an error after the deposit but before the withdrawal? The customer's accounts will have too much money, and the bank's board of directors will fire the entire IT staff. The point is, no matter how you order the operations, the accounts will be left in an inconsistent state if one operation is committed and the other is not. You need to guarantee that either both operations occur or that neither does. That's what transaction management is all about.

The default for a database connection is autocommit; that is, each executed statement is automatically committed to the database. Thus, for transaction management you first need to turn off autocommit for the connection by calling `setAutoCommit(false)`.

Typically, you use a `try/catch/finally` block to properly handle the transaction management. First, you should record the autocommit status. Then, in the `try` block, you should call `setAutoCommit(false)` and execute a set of queries or updates. If a failure occurs, you call `rollback` in the `catch` block; if the transactions are successful, you call `commit` at the end of the `try` block. Either way, you reset the autocommit status in the `finally` block.

Following is a template for this transaction management approach.

```
Connection connection =
  DriverManager.getConnection(url, username, password);
boolean autoCommit = connection.getAutoCommit();
Statement statement;
try {
  connection.setAutoCommit(false);
  statement = connection.createStatement();
  statement.execute(...);
  statement.execute(...);
  ...
  connection.commit();
} catch(SQLException sqle) {
  connection.rollback();
} finally {
  statement.close();
  connection.setAutoCommit(autoCommit);
}
```

Here, the statement for obtaining a connection from the `DriverManager` is outside the `try/catch` block. That way, `rollback` is not called unless a connection is successfully obtained. However, the `getConnection` method can still throw an `SQLException` and must be thrown by the enclosing method or be caught in a separate `try/catch` block.

In Listing 17.13, we add new recordings to the `music` table as a transaction block (see Section 18.5 to create the `music` table). To generalize the task, we create a `TransactionBean`, Listing 17.14, in which we specify the connection to the database and submit a block of SQL statements as an array of strings. The bean then loops through the array of SQL statements, executes each one of them, and if an `SQLException` is thrown, performs a `rollback` and rethrows the exception.

Listing 17.13 Transactions.java

```
package coreservlets;

import java.sql.*;
import coreservlets.beans.*;

/** An example to demonstrate submission of a block of
 *  SQL statements as a single transaction. Specifically,
 *  four new records are inserted into the music table.
 *  Performed as a transaction block so that if a problem
 *  occurs, a rollback is performed and no changes are
 *  committed to the database.
 */

public class Transactions {
  public static void main(String[] args) {
    if (args.length < 5) {
      printUsage();
      return;
    }
    String vendor = args[4];
    // Change to DriverUtilities2.loadDrivers() to load
    // vendor drivers from an XML file instead of loading
    // hard-coded vendor drivers in DriverUtilities.
    DriverUtilities.loadDrivers();
    if (!DriverUtilities.isValidVendor(vendor)) {
      printUsage();
      return;
    }
    String driver = DriverUtilities.getDriver(vendor);
    String host = args[0];
    String dbName = args[1];
    String url =
      DriverUtilities.makeURL(host, dbName, vendor);
    String username = args[2];
    String password = args[3];
    doTransactions(driver, url, username, password);
  }

  private static void doTransactions(String driver,
                                     String url,
                                     String username,
                                     String password) {
```

Listing 17.13 Transactions.java *(continued)*

```
    String[] transaction =
    { "INSERT INTO music VALUES " +
      " ( 9, 'Chopin',       'No. 2 in F minor',  100, 17.99)",
      "INSERT INTO music VALUES " +
      " (10, 'Tchaikovsky',  'No. 1 in Bb minor', 100, 24.99)",
      "INSERT INTO music VALUES " +
      " (11, 'Ravel',        'No. 2 in D major',  100, 14.99)",
      "INSERT INTO music VALUES " +
      " (12, 'Schumann',     'No. 1 in A minor',  100, 14.99)"};
    TransactionBean bean = new TransactionBean();
    try {
      bean.setConnection(driver, url, username, password);
      bean.execute(transaction);
    } catch (SQLException sqle) {
      System.err.println("Transaction failure: " + sqle);
    } finally {
      bean.close();
    }
  }

  private static void printUsage() {
    System.out.println("Usage: Transactions host " +
                       "dbName username password " +
                       "vendor.");
  }
}
```

Listing 17.14 TransactionBean.java

```
package coreservlets.beans;

import java.io.*;
import java.sql.*;
import java.util.*;
import coreservlets.*;

/** Bean for performing JDBC transactions. After specifying
 *  the connection, submit a block of SQL statements as a
 *  single transaction by calling execute. If an SQLException
 *  occurs, any prior statements are automatically rolled back.
 */
```

Listing 17.14 TransactionBean.java *(continued)*

```
public class TransactionBean {
  private Connection connection;

  public void setConnection(Connection connection) {
    this.connection = connection;
  }

  public void setConnection(String driver, String url,
                            String username, String password) {
    setConnection(ConnectionInfoBean.getConnection(
                    driver, url, username, password));
  }

  public Connection getConnection() {
    return(connection);
  }

  public void execute(List list) throws SQLException {
    execute((String[])list.toArray(new String[list.size()]));
  }

  public void execute(String transaction)
      throws SQLException {
    execute(new String[] { transaction });
  }

  /** Execute a block of SQL statements as a single
   *  transaction.  If an SQLException occurs, a rollback
   *  is attempted and the exception is thrown.
   */

    public void execute(String[] transaction)
      throws SQLException {
    if (connection == null) {
      throw new SQLException("No connection available.");
    }
    boolean autoCommit = connection.getAutoCommit();
    try {
      connection.setAutoCommit(false);
      Statement statement = connection.createStatement();
      for(int i=0; i<transaction.length; i++) {
        statement.execute(transaction[i]);
      }
      statement.close();
    } catch(SQLException sqle) {
      connection.rollback();
      throw sqle;
```

Listing 17.14 TransactionBean.java *(continued)*

```
    } finally {
      connection.commit();
      connection.setAutoCommit(autoCommit);
    }
  }

  public void close() {
    if (connection != null) {
      try {
        connection.close();
      } catch(SQLException sqle) {
        System.err.println(
          "Failed to close connection: " + sqle);
      } finally {
        connection = null;
      }
    }
  }
}
```

The preceding example demonstrates the use of a bean for submitting transactions to a database. This approach is excellent for a servlet; however, in a JSP page, you may want to use the `sql:transaction` action available in the JSP Standard Tag Library (JSTL). See Volume 2 of this book for details on JSTL.

17.7 Mapping Data to Objects by Using ORM Frameworks

Because of the need to easily move data back and forth from a database to a Java object, numerous vendors have developed frameworks for mapping objects to relational databases. This is a powerful capability since object-oriented programming and relational databases have always had an impedance mismatch: objects understand both state and behavior and can be traversed through relationships with other objects, whereas relational databases store information in tables but are typically related through primary keys.

Table 17.1 summarizes a few popular object-to-relational mapping (ORM) frameworks. Numerous other ORM frameworks are available. For a comparison of Java-based ORM products, see http://c2.com/cgi-bin/wiki?ObjectRelationalTool-

Comparison. Another excellent source for ORM frameworks and tutorials is located at http://www.javaskyline.com/database.html. (Remember that the book's source code archive at http://www.coreservlets.com/ contains up-to-date links to all URLs mentioned in the book.)

Table 17.1 Object-to-Relational Mapping Frameworks

Framework	URL
Castor	http://castor.exolab.org/
CocoBase	http://www.cocobase.com/
FrontierSuite	http://www.objectfrontier.com/
Kodo JDO	http://www.solarmetric.com/
ObJectRelationalBridge	http://db.apache.org/ojb/
TopLink	http://otn.oracle.com/products/ias/toplink/

Many, but not all, of these frameworks support the API for Java Data Objects (JDO). The JDO API provides a complete object-oriented approach to manage objects that are mapped to a persistent store (database). Detailed coverage of ORM frameworks and JDO is beyond the scope of this book. However, we provide a brief summary to show the power that JDO provides. For more information on JDO, see the online material at http://java.sun.com/products/jdo/ and http://jdocentral.com/. In addition, you can refer to *Core Java Data Objects* by Sameer Tyagi et al.

In JDO frameworks, the developer must provide XML-based metadata for each Java class that maps to the relational database. The metadata defines the persistent fields in each class and the potential role of each field relative to the database (e.g., the primary key). Once the metadata and source code for each Java class are defined, the developer must run a framework utility to generate the necessary JDO code to support persistence of the object's fields in the persistent store (database).

JDO framework utilities take one of two approaches to modify a Java class in support of the persistent store: the first approach is to modify the source code before compiling, the second approach is to modify the bytecode (.class file) after compiling the source code. The second approach is more common because it simplifies the maintenance of the source code—the generated database code is never seen by the developer.

In the case of a JDO implementation for an SQL database, the framework utility generates all the necessary code required by the JDO persistence manager to `INSERT` new rows in the database, as well as to perform `UPDATE` and `DELETE` operations for persisting modifications to the data. The developer is not required to write

any SQL or JDBC code; the framework utility generates all the necessary code, and the persistence manager generates all the necessary communication with the database. Once the framework is set up, the developer simply needs to create objects and understand the JDO API.

Listing 17.15 shows Music.jdo, an example of how metadata for the `Music` class (Listing 17.16) is defined for SolarMetric's Kodo JDO implementation. The XML file Music.jdo maps the `Music` class to a table in the database by using an `extension` element with a `key` attribute of `table` and a `value` attribute of `music`. It is not necessary that the database already have a table named `music`; in fact, the Kodo framework creates all tables necessary in the database for the persistent storage, possibly using modified table names. The `name` attribute simply defines a mapping for the framework.

The .jdo file further designates a `field` element for each field in the class that must be persisted in the database. Each `field` element defines an `extension` element to map a field in the class to a column in the database table, where the `value` attribute clarifies the name of the database column. The `extension` elements are vendor specific, so be sure to consult the documentation of your JDO vendor for the proper values for the `key` attribute.

The `PersistenceManager` class provides access to the persistent store (database). For example, Listing 17.17 shows how to insert fields associated with new objects into the persistent store with the `makePersistentAll` method. Changes to the persistent store are managed as a transaction and must be placed between calls to the `begin` and `commit` methods of the `Transaction` class. Thus, to insert the fields associated with a `Music` object into the database, you simply call the appropriate `set`*`Xxx`* methods on the `Music` object and then invoke the `makePersistentAll` method within a transaction. The JDO persistence manager automatically creates and executes the SQL statements to commit the data to the database. In a similar manner, the deletion of fields associated with a `Music` object is handled through the `makeDeletePersistent` method of `PersistenceManager`. For more complicated interaction with the persistent store, JDO provides a `Query` class to execute queries and return the results as a `Collection` of objects.

Lastly, the location of the database, the username, the password, and other system-specific information is read from system properties (specified by Listing 17.18 in this example).

Listing 17.15 Music.jdo

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
"-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN"
"http://java.sun.com/dtd/jdo_1_0.dtd">
<jdo>
  <package name="coreservlets.jdo">
    <class name="Music" >
      <extension vendor-name="kodo"
                 key="table" value="music"/>
      <extension vendor-name="kodo"
                 key="lock-column" value="none"/>
      <extension vendor-name="kodo"
                 key="class-column" value="none"/>
      <field name="id" primary-key="true">
        <extension vendor-name="kodo"
                   key="data-column" value="id"/>
      </field>
      <field name="composer">
        <extension vendor-name="kodo"
                   key="data-column" value="composer"/>
      </field>
      <field name="concerto">
        <extension vendor-name="kodo"
                   key="data-column" value="concerto"/>
      </field>
      <field name="available">
        <extension vendor-name="kodo"
                   key="data-column" value="available"/>
      </field>
      <field name="price">
        <extension vendor-name="kodo"
                   key="data-column" value="price"/>
      </field>
    </class>
  </package>
</jdo>
```

Listing 17.16 Music.java

```
package coreservlets.jdo;

 /** Music object corresponding to a record in a database.
  *  A Music object/record provides information about
  *  a concerto that is available for purchase and
  *  defines fields for the ID, composer, concerto,
  *  items available, and sales price.
  */

public class Music {
  private int id;
  private String composer;
  private String concerto;
  private int available;
  private float price;

  public Music() { }

  public Music(int id, String composer, String concerto,
               int available, float price) {
    setId(id);
    setComposer(composer);
    setConcerto(concerto);
    setAvailable(available);
    setPrice(price);
  }

  public void setId(int id) {
    this.id = id;
  }

  public int getId() {
    return(id);
  }

  public void setComposer(String composer) {
    this.composer = composer;
  }

  public String getComposer() {
    return(concerto);
  }

  public void setConcerto(String concerto) {
    this.concerto = concerto;
  }
```

Listing 17.16 Music.java *(continued)*

```
  public String getConcerto() {
    return(composer);
  }

  public void setAvailable(int available) {
    this.available = available;
  }

  public int getAvailable() {
    return(available);
  }

  public void setPrice(float price) {
    this.price = price;
  }

  public float getPrice() {
    return(price);
  }
}
```

Listing 17.17 PopulateMusicTable.java

```
package coreservlets.jdo;

import java.util.*;
import java.io.*;
import javax.jdo.*;

/** Populate database with music records by using JDO.
 */
public class PopulateMusicTable {
  public static void main(String[] args) {
    // Create seven new music objects to place in the database.
    Music[] objects = {
      new Music(1, "Mozart", "No. 21 in C# minor", 7, 24.99F),
      new Music(2, "Beethoven", "No. 3 in C minor", 28, 10.99F),
      new Music(3, "Beethoven", "No. 5 Eb major", 33, 10.99F),
      new Music(4, "Rachmaninov", "No. 2 in C minor", 9, 18.99F),
      new Music(5, "Mozart", "No. 24 in C minor", 11, 21.99F),
      new Music(6, "Beethoven", "No. 4 in G", 33, 12.99F),
      new Music(7, "Liszt", "No. 1 in Eb major", 48, 10.99F)
    };
```

Listing 17.17 PopulateMusicTable.java *(continued)*

```
    // Load properties file with JDO information. The properties
    // file contains ORM Framework information specific to the
    // vendor and information for connecting to the database.
    Properties properties = new Properties();
    try {
      FileInputStream fis =
        new FileInputStream("jdo.properties");
      properties.load(fis);
    } catch(IOException ioe) {
      System.err.println("Problem loading properties file: " +
                         ioe);
      return;
    }

    // Initialize manager for persistence framework.
    PersistenceManagerFactory pmf =
      JDOHelper.getPersistenceManagerFactory(properties);
    PersistenceManager pm = pmf.getPersistenceManager();

    // Write the new Music objects to the database.
    Transaction transaction = pm.currentTransaction();
    transaction.begin();
    pm.makePersistentAll(objects);
    transaction.commit();
    pm.close ();
  }
}
```

Listing 17.18 jdo.properties

```
# Configuration information for Kodo JDO Framework and
# MySQL database.
javax.jdo.PersistenceManagerFactoryClass=
  com.solarmetric.kodo.impl.jdbc.JDBCPersistenceManagerFactory
javax.jdo.option.RetainValues=true
javax.jdo.option.RestoreValues=true
javax.jdo.option.Optimistic=true
javax.jdo.option.NontransactionalWrite=false
javax.jdo.option.NontransactionalRead=true
javax.jdo.option.Multithreaded=true
javax.jdo.option.MsWait=5000
javax.jdo.option.MinPool=1
javax.jdo.option.MaxPool=80
```

Listing 17.18 jdo.properties *(continued)*

```
javax.jdo.option.IgnoreCache=false
javax.jdo.option.ConnectionUserName: brown
javax.jdo.option.ConnectionURL: jdbc:mysql://localhost/csajsp
javax.jdo.option.ConnectionPassword: larry
javax.jdo.option.ConnectionDriverName: com.mysql.jdbc.Driver
com.solarmetric.kodo.impl.jdbc.WarnOnPersistentTypeFailure=true
com.solarmetric.kodo.impl.jdbc.SequenceFactoryClass=
  com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory
com.solarmetric.kodo.impl.jdbc.FlatInheritanceMapping=true
com.solarmetric.kodo.EnableQueryExtensions=false
com.solarmetric.kodo.DefaultFetchThreshold=30
com.solarmetric.kodo.DefaultFetchBatchSize=10
com.solarmetric.kodo.LicenseKey=5A8A-D98C-DB5F-6070-6000
```